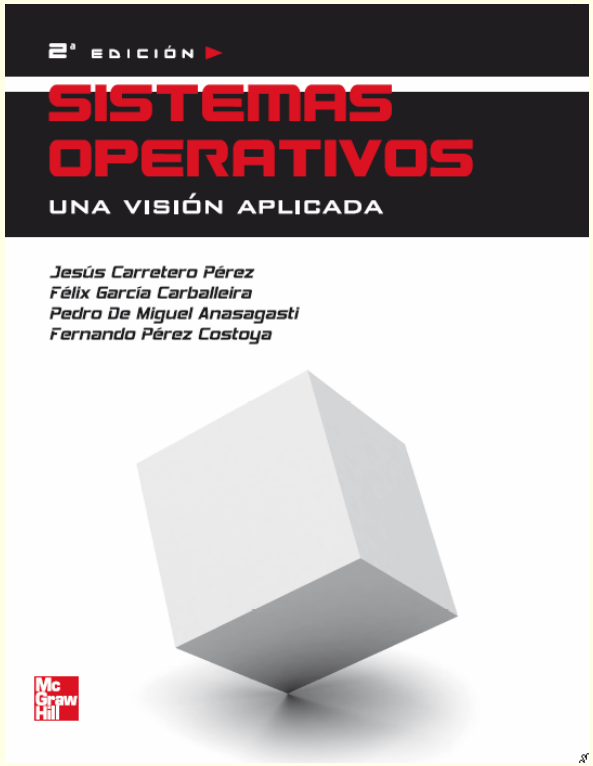
# Sistemas operativos
## 2ª edición




# Capítulo 5
# Gestión de memoria
## (extracto de las transparencias del libro)

# Contenido

- Aspectos generales de la gestión de memoria
- Modelo de memoria de un proceso
- Esquemas de gestión de la memoria del sistema
- Memoria virtual

# Aspectos generales de la gestión de memoria

- SO multiplexa recursos entre procesos
  - Gestión de procesos: Reparto (espacial) de procesador
  - Gestión de memoria: Reparto (espacial y temporal) de memoria
- Gestión integral de la memoria: no sólo SO
  - Compilador, montador y hardware de gestión de memoria
  - Presentación se centra en SO (Libro ofrece visión integral)
- Gestor de memoria: elevada complejidad y dependencia del HW
- El largo camino desde el acceso simbólico al real
  - Código de programa hace referencia a variables y funciones
  - En tiempo de ejecución programa genera accesos a memoria
  - Requiere varias etapas de transformación
    - compilador, montador, SO y HW se reparten el trabajo

# Niveles de gestión de memoria

- **Nivel de procesos**
  - Reparto de memoria del sistema entre procesos
- **Nivel de regiones**
  - Reparto de memoria del proceso entre regiones
- **Nivel de zonas** (si aplicable)
  - Reparto de espacio de región entre sus zonas
    - No gestionado por sistema operativo
    - Por ejemplo, región de *heap*
- Cada nivel es caso del problema gral. de la asignación de espacio
  - En cada nivel se puede usar esquema diferente
  - Un nivel reparte espacio proporcionado por nivel superior

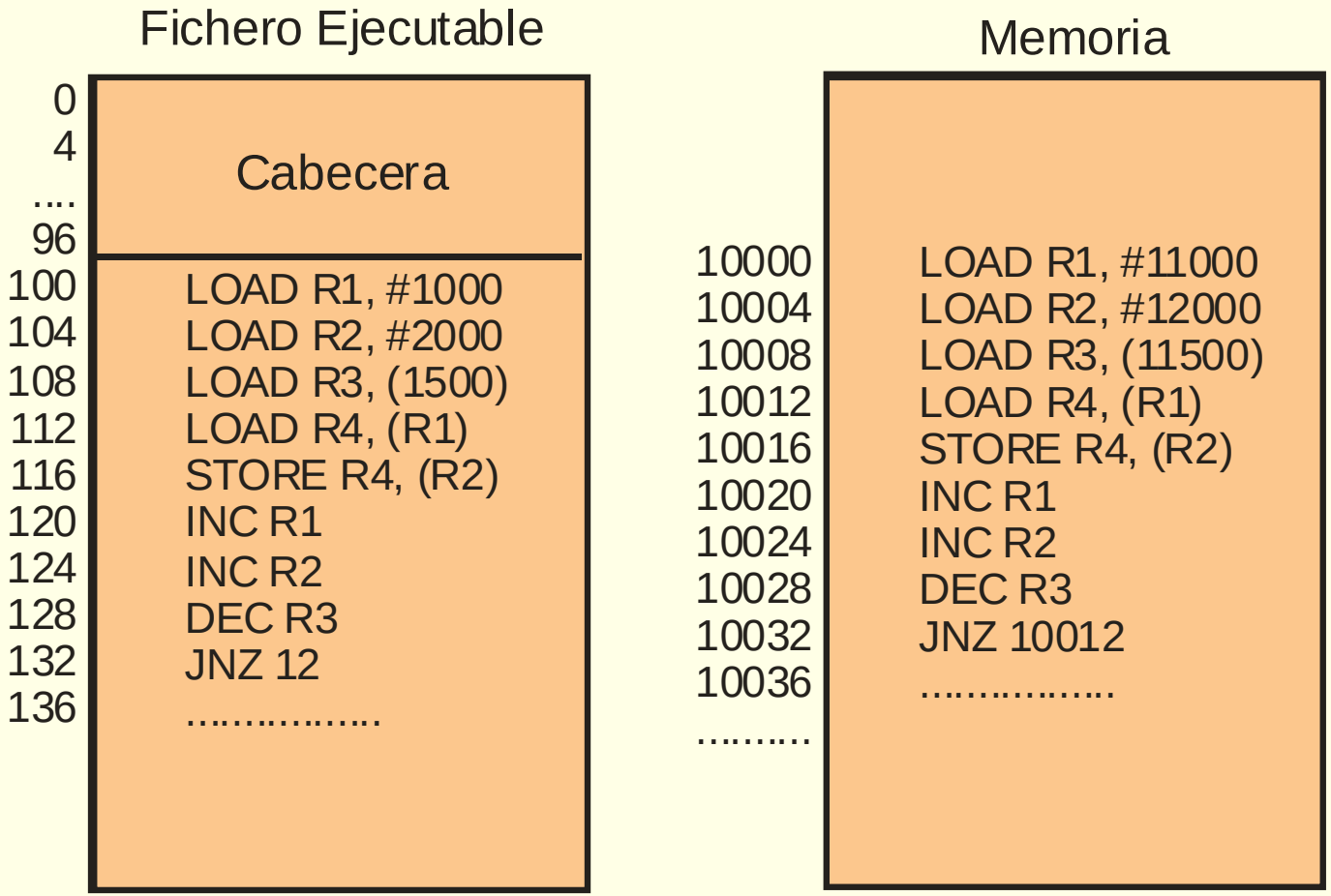
# Objetivos del sistema de gestión de memoria

- Necesidades de los programas y del SO
  - Espacios lógicos independientes
  - Protección
  - Compartir memoria
  - Aprovechamiento del espacio de memoria
  - Soporte de regiones

# Espacio lógico independependientes y protección

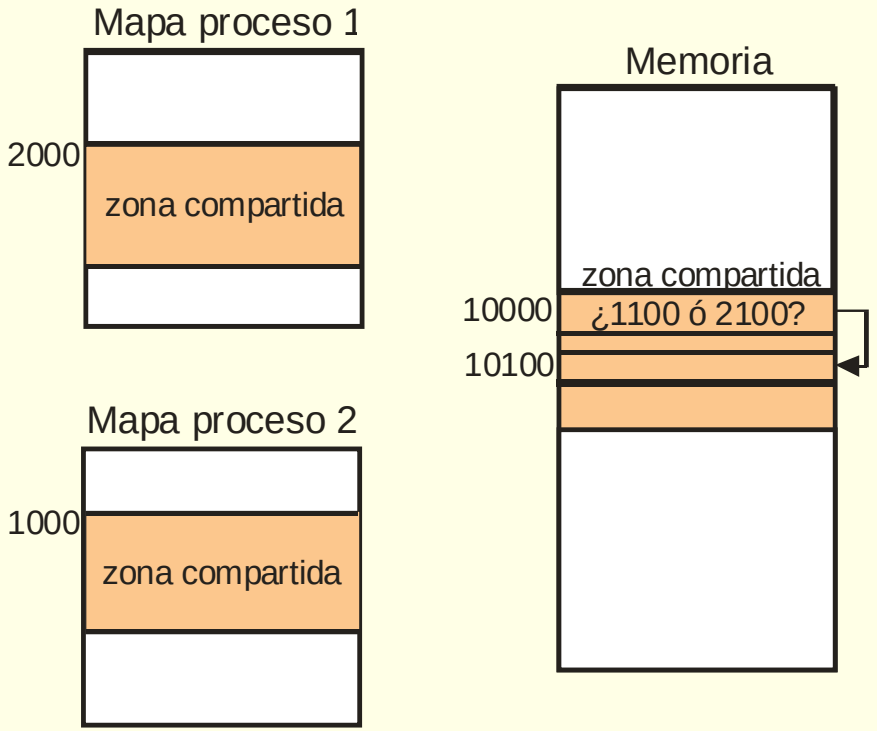
- Código en ejecutable incluye referencias entre 0 y N
  - En SO multiprogramado: necesidad de reubicar
- **Reubicar**: Traducir direcciones lógicas a físicas
  - Crea espacio lógico independiente para proceso
  - Provee protección: aislamiento del SO y de procesos entre sí
- Dos alternativas:
  - **Reubicación software**: previa a la ejecución del proceso
  - **Reubicación hardware**: en tiempo de ejecución por MMU
    - Posibilita protección
- ¿Es necesaria reubicación de direcciones en SO?
  - Uso de reubicación proporciona más flexibilidad
- SO no sólo requiere utilizar su mapa sino toda la memoria
  - Además, necesita "ver" espacio lógico de cada proceso

# Reubicación software en la carga



# Compartimiento de memoria para comunicación

- Dir. lógicas de procesos corresponden con misma dir. física




**Problema de las autorreferencias**

# Compartimiento de memoria para optimizar su uso

- Compartir memoria permite mejor aprovechamiento
  - Compartir código de programas o de bibliotecas
- Datos de programa y de biblioteca no deben compartirse
  - Pero inicialmente idénticos
  - Diferir copia de cada dato hasta que se modifique (COW)
  - También aplicable a `fork`

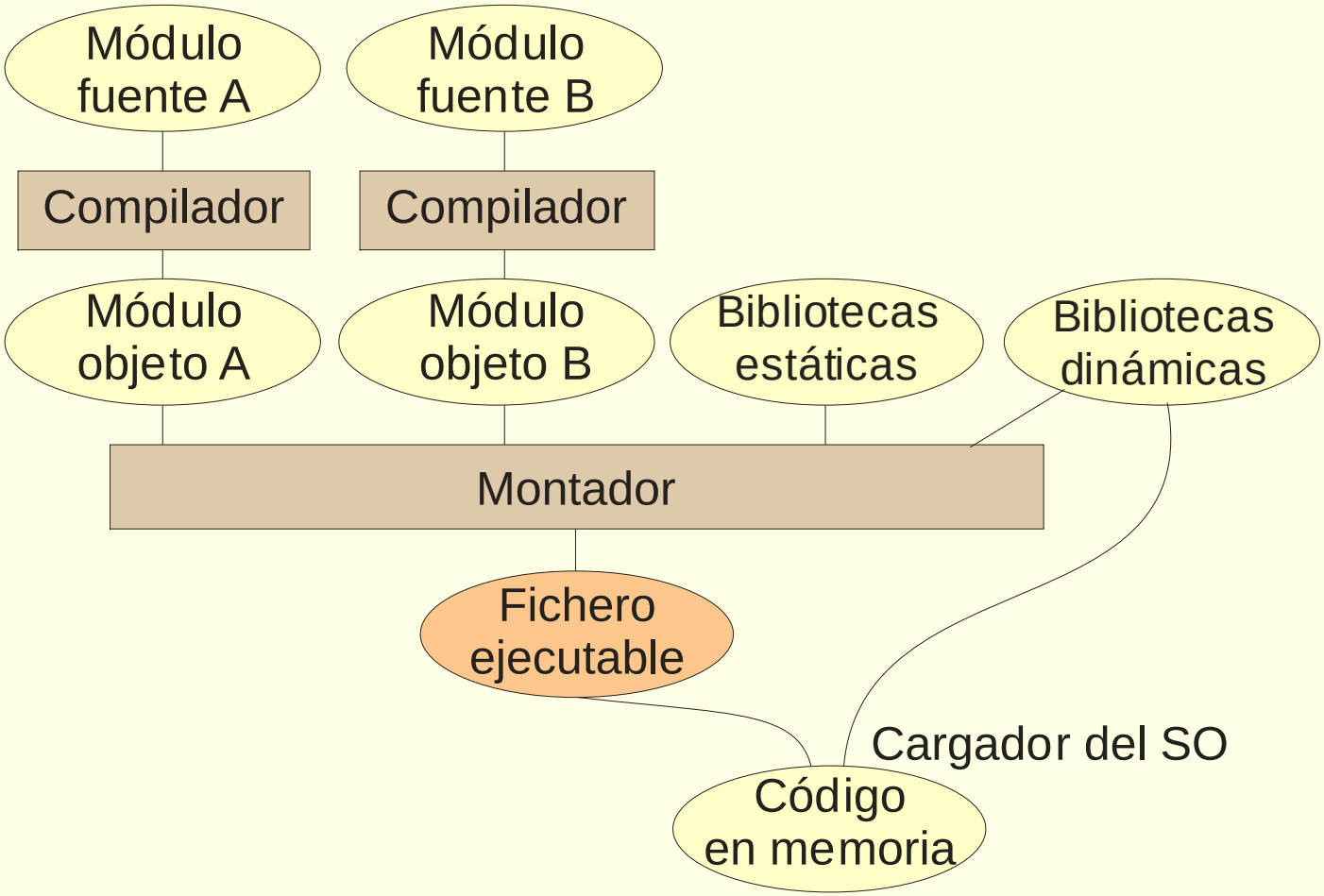
# Buen aprovechamiento de la memoria

- Todo byte debería almacenar información de utilidad pero...
  - Desperdicio inherente a la propia gestión (**fragmentación**)
    - Externa o interna
  - Gasto de propia gestión de memoria (estructuras de datos)
- Mejor aprovechamiento → mayor grado de multiprogramación
  → mejor rendimiento
- Para mejorar rendimiento, uso de **memoria virtual**
  - Requisito de todo SO de propósito general

# Soporte de regiones

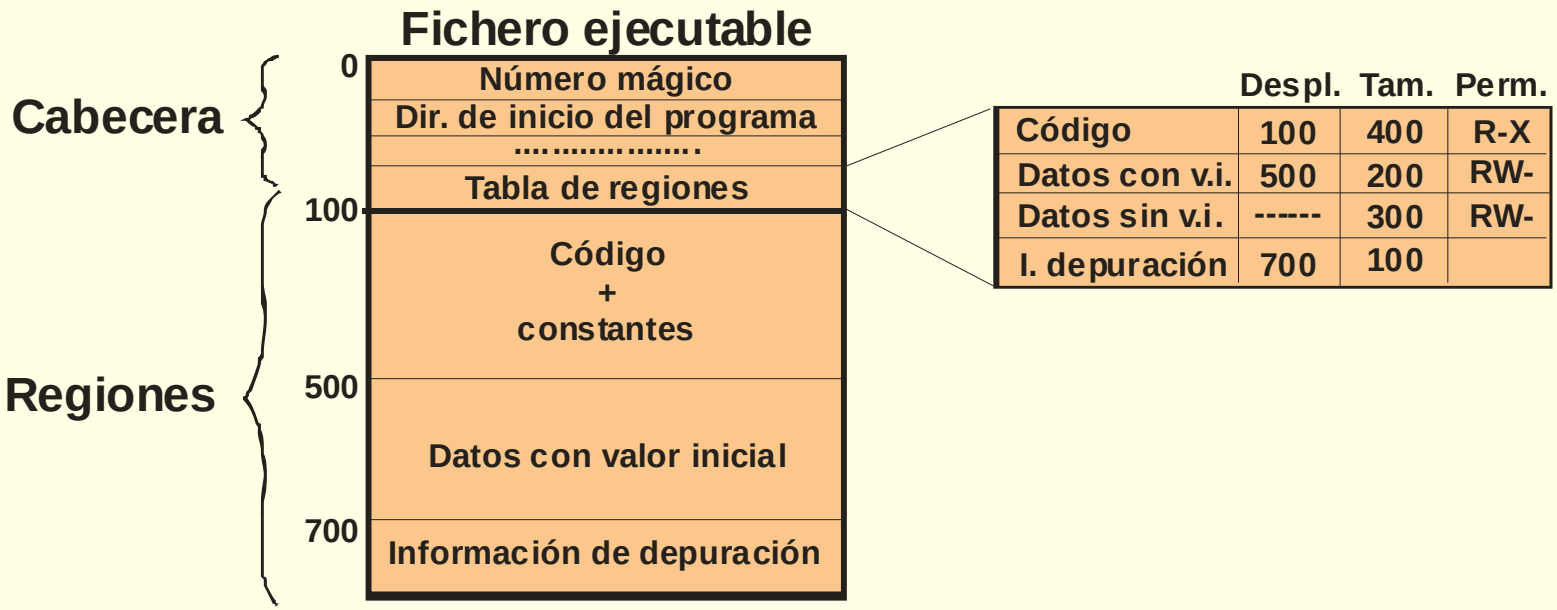
- Mapa de proceso no homogéneo
  - Conjunto de regiones con distintas características
    - P.e. permiso de acceso
      - Detección de operaciones no permitidas
- Mapa de proceso dinámico
  - Regiones cambian de tamaño, se crean y destruyen
  - Huecos en el mapa del proceso
    - No deberían consumir espacio
- Aplicaciones con espacio de direcciones disperso
  - Muchas regiones algunas relativamente pequeñas

# Modelo memoria de proceso: ciclo vida de programa



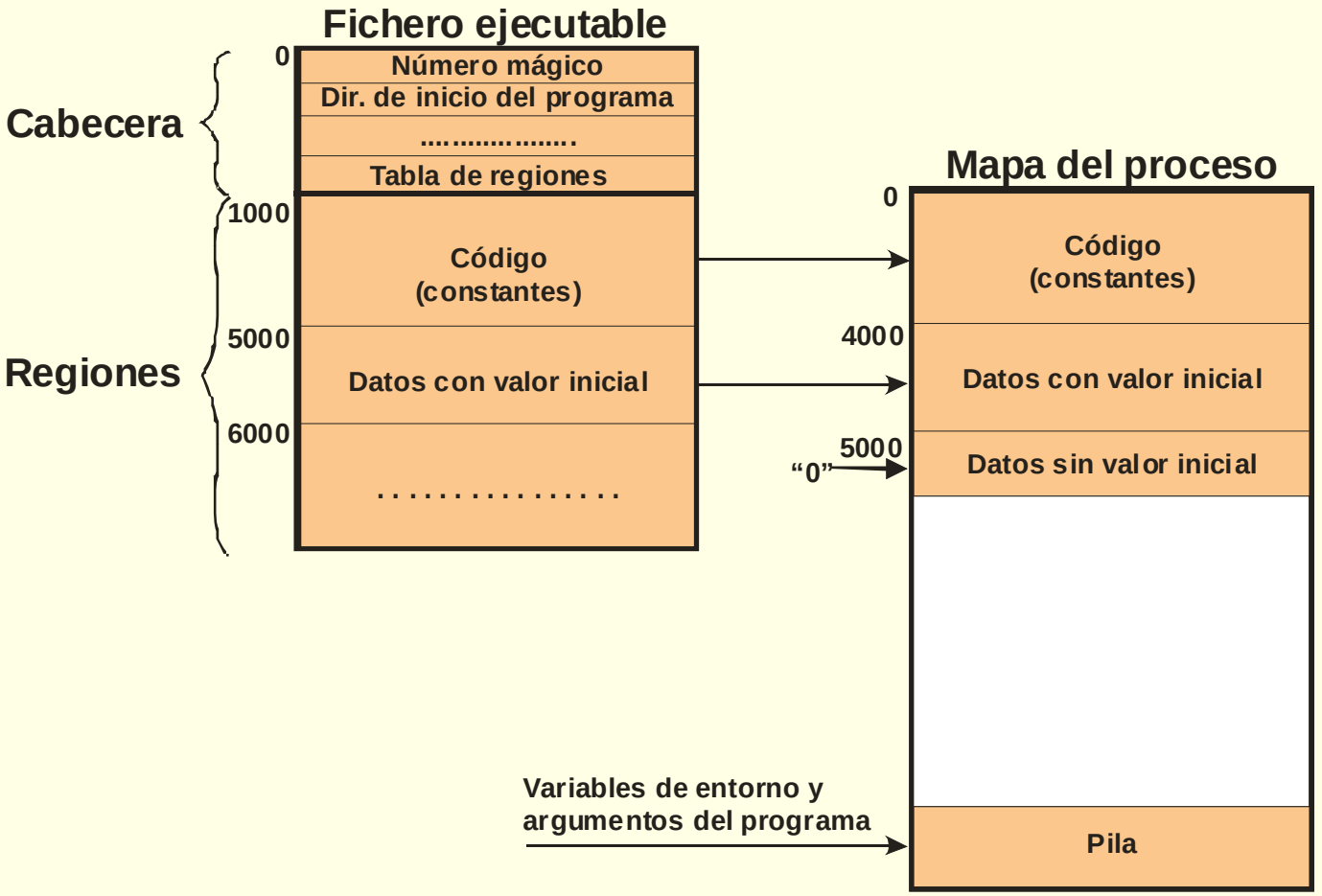
# Formato del ejecutable



# Mapa de memoria de un proceso

- Evolución del mapa de memoria del proceso: Nivel de regiones
- Mapa de memoria o imagen del proceso: conjunto de regiones
  - Uso de tabla de regiones
- Región: zona contigua tratada como unidad
  - dirección de comienzo y tamaño inicial
  - soporte: En fichero o sin soporte (anónima)
  - protección: RWX; compartida o privada; tamaño fijo o variable
- Ejecución de programa: Crea mapa a partir de ejecutable
  - Regiones de mapa inicial → Regiones de ejecutable + pila
  - MMU realiza reubicación de dir. lógica de proceso → dir. física

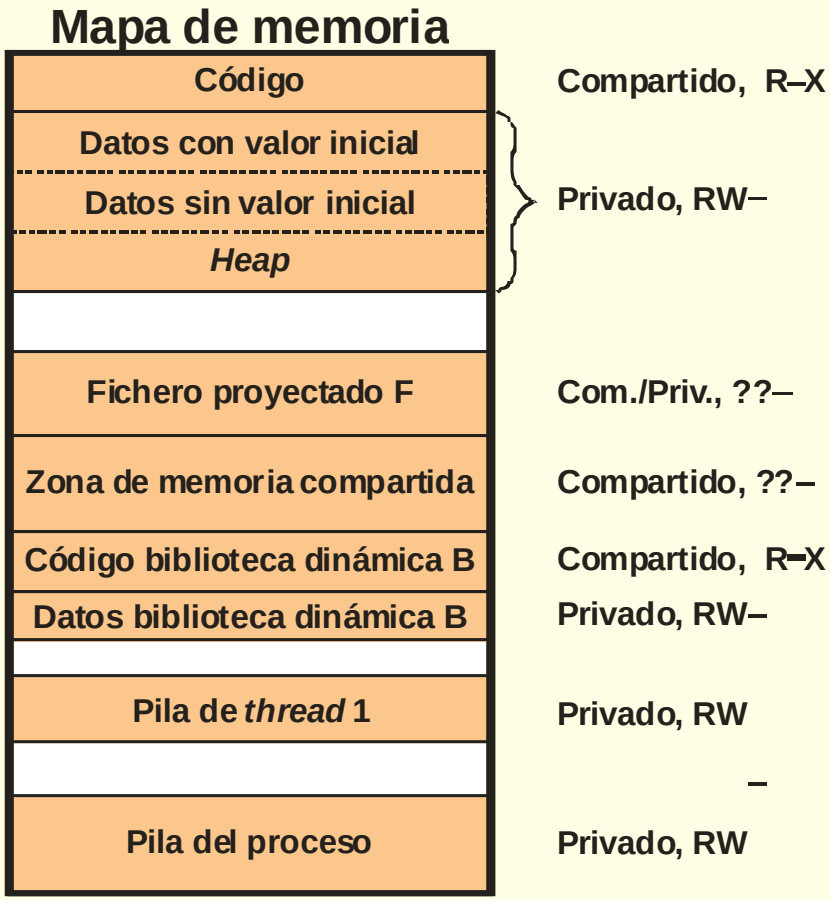
# Crear mapa desde ejecutable



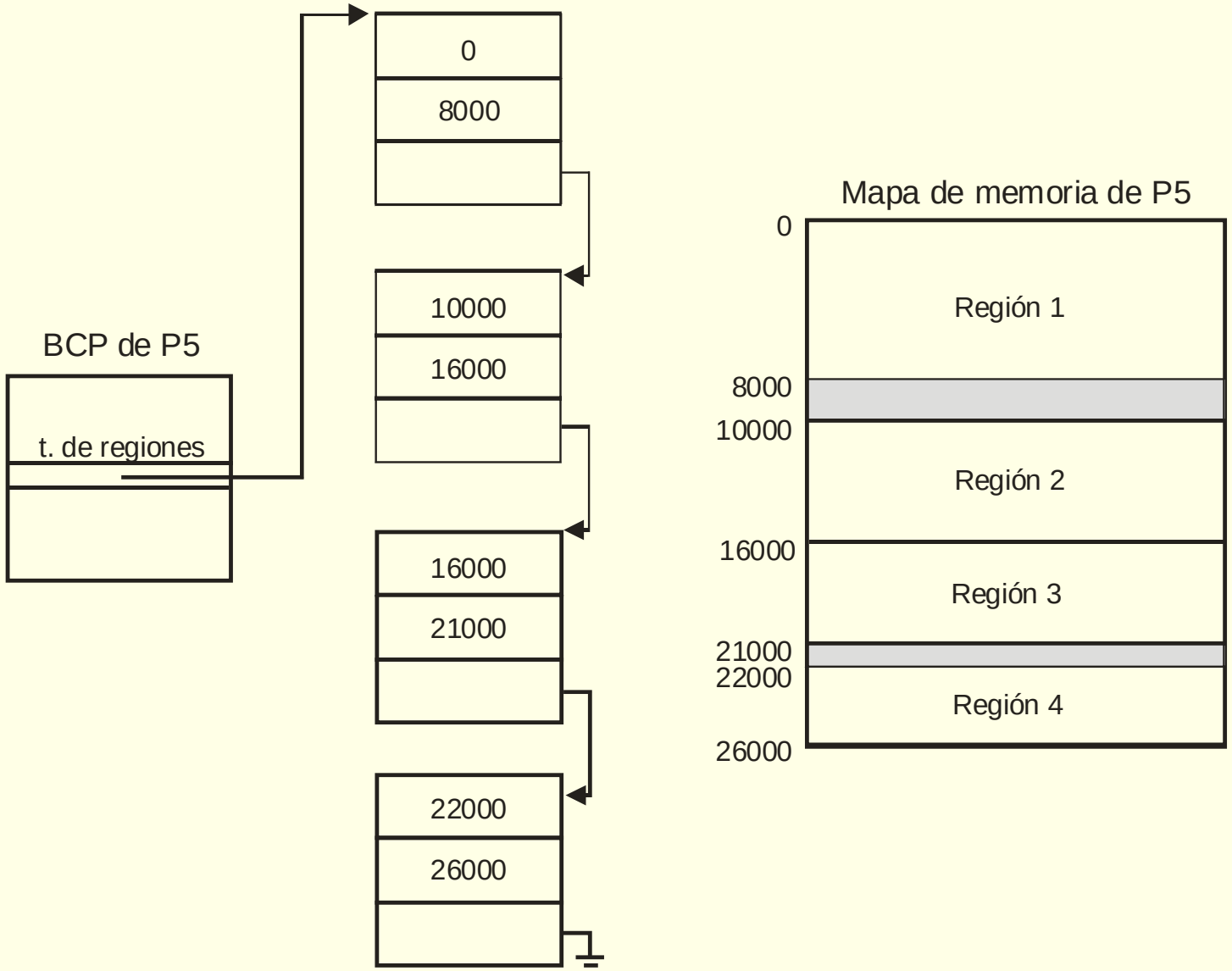
# Regiones del mapa de memoria

| Región | Soporte | Protección | Comp/Priv | Tam | Ubicación |
|---|---|---|---|---|---|
| Código | Fichero | RX | Compartida | Fijo | Prefijada |
| Dat. con v.i. | Fichero | RW | Privada | Fijo | Prefijada |
| Dat. sin v.i. | Sin soporte | RW | Privada | Fijo | Prefijada |
| Pilas | Sin soporte | RW | Privada | Var | Cualquiera |
| Heap | Sin soporte | RW | Privada | Var | Cualquiera |
| F. Proyect. | Fichero | por usuario | Comp./Priv. | Fijo | Cualquiera |
| M. Comp. | Sin soporte | por usuario | Compartida | Fijo | Cualquiera |
| Bib.dinám. | Regiones para código y datos (con y sin valor inicial) | | | | |

# Mapa de memoria de un proceso hipotético



# Implementación de la tabla de regiones

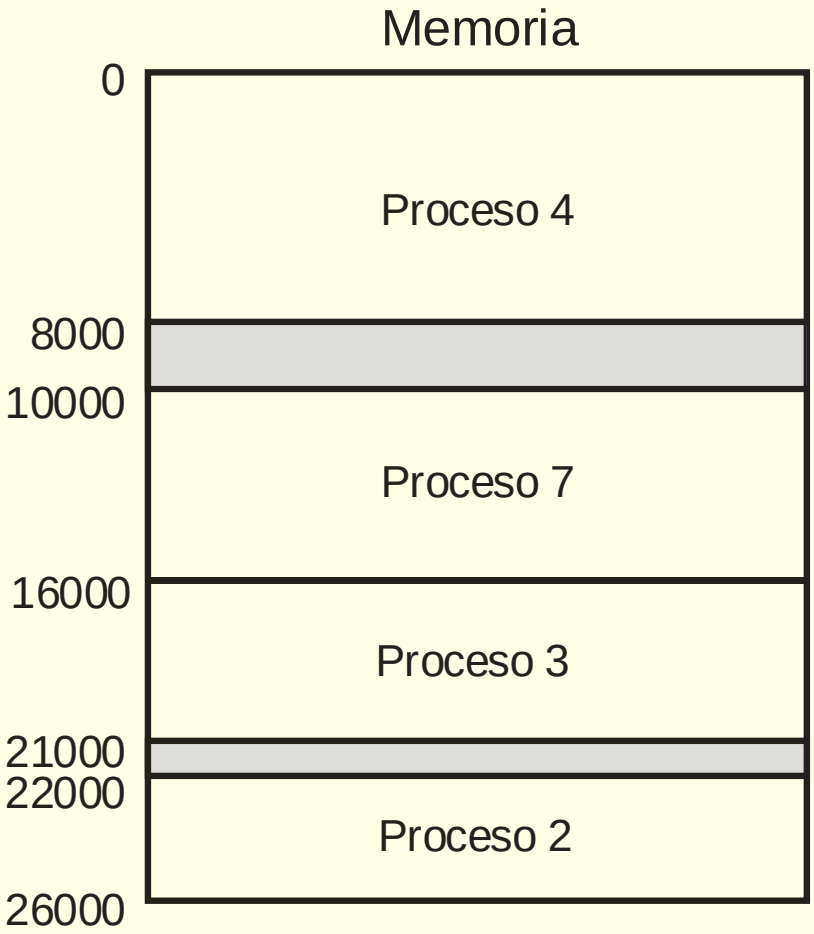


# Esquemas de gestión de la memoria del sistema

- Nivel de procesos
  - ¿Cómo se reparte la memoria entre mapas de los procesos?
  - Muy ligado al hardware de gestión de memoria
- Esquemas de gestión analizados:
  - Asignación contigua (obsoleto en SO propósito general)
  - Segmentación (obsoleto en SO propósito general)
  - Paginación: soporte mínimo esperado por SO propósito general
  - Segmentación paginada
    - Más sofisticado que paginación pero por portabilidad
      - Mayoría de SSOO propósito general desactivan la segmentación
- Recordatorio de objetivos del sistema de gestión de memoria:
    - Espacios lógicos independientes, soporte regiones, compartir, buen aprovechamiento y soporte memoria virtual

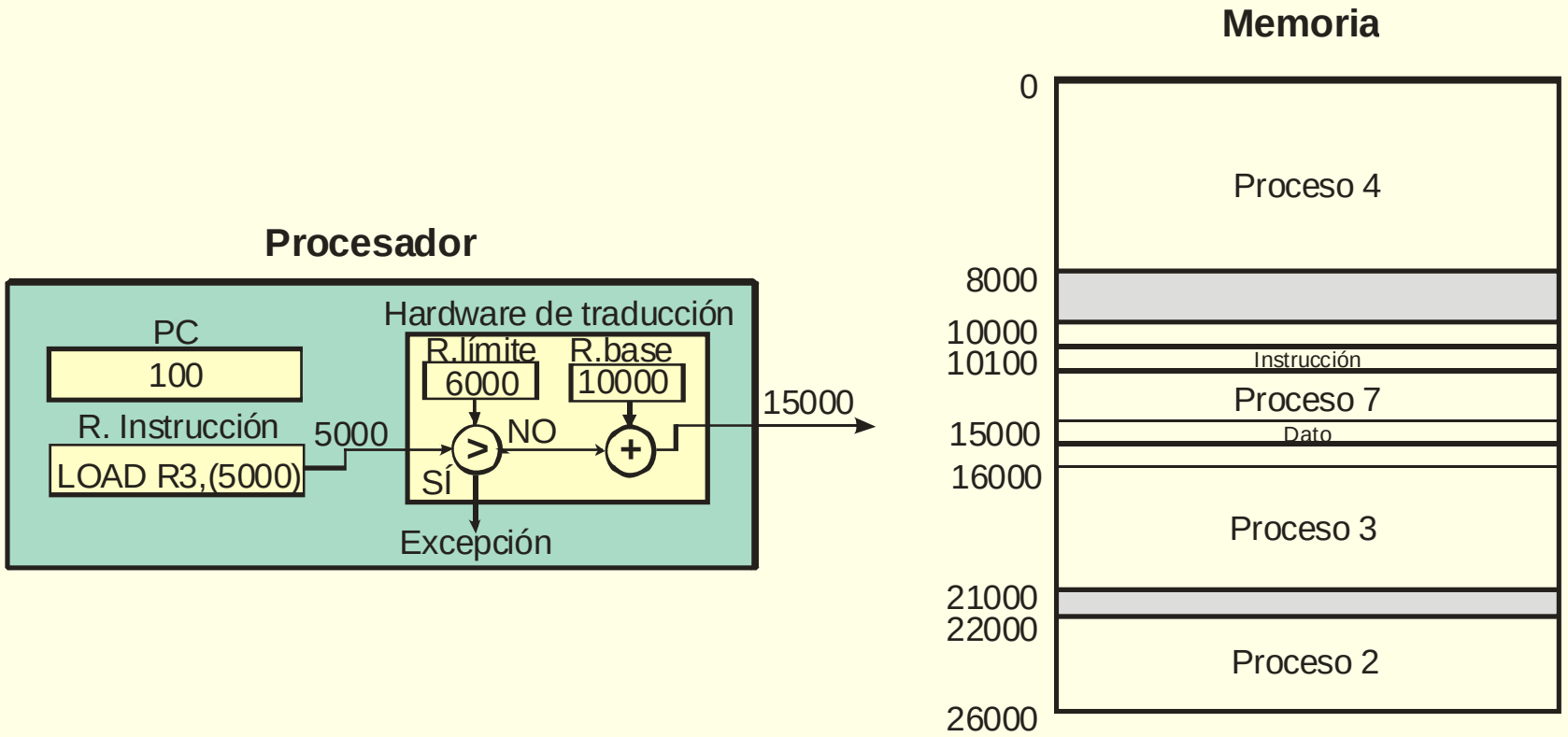
# Asignación contigua




- Deficiencias de asignación contigua
  - No soporte regiones, no compartir, no base de m. virtual

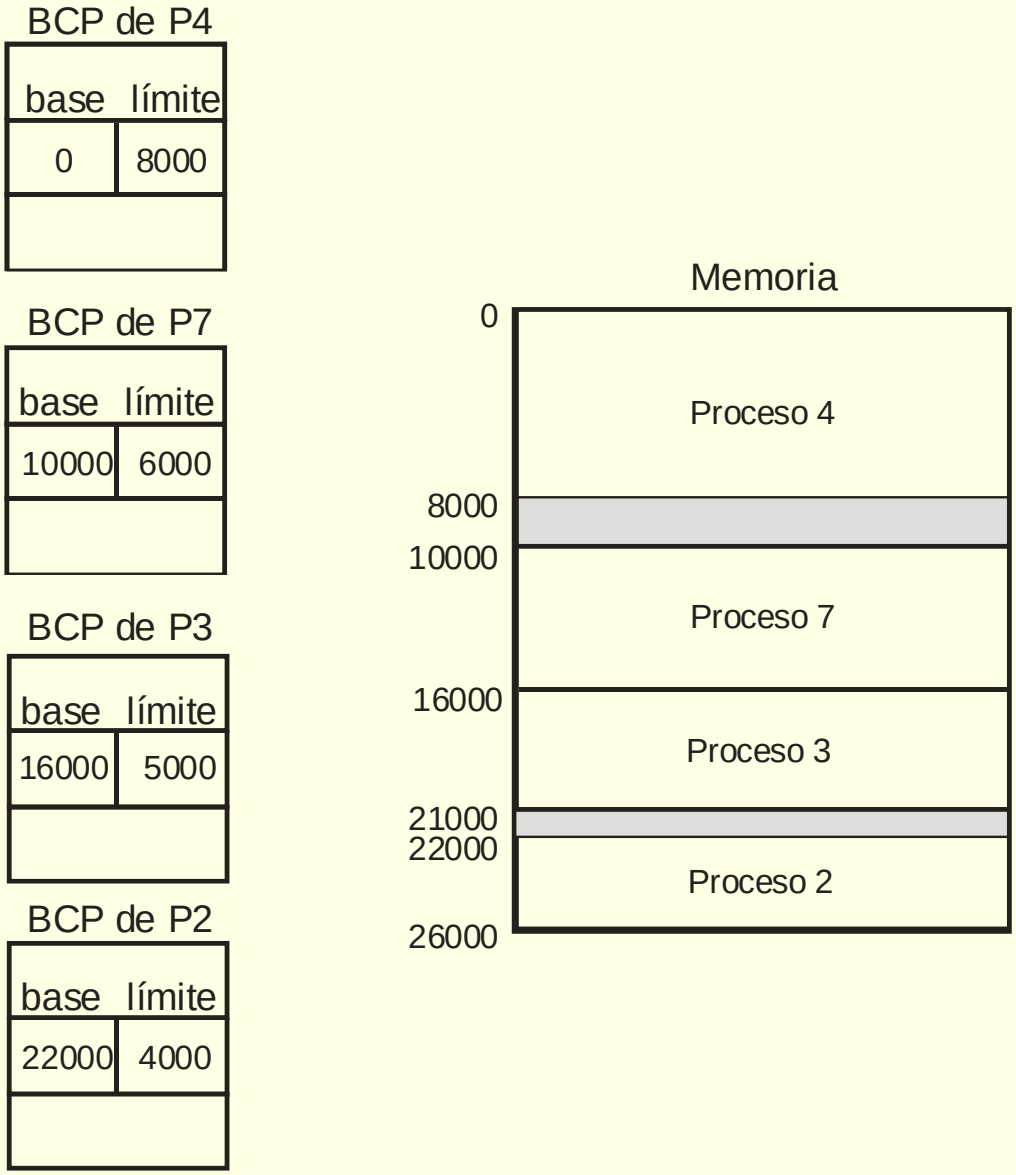
# Registros base y límite

- Protección + reubicación de procesos por hardware



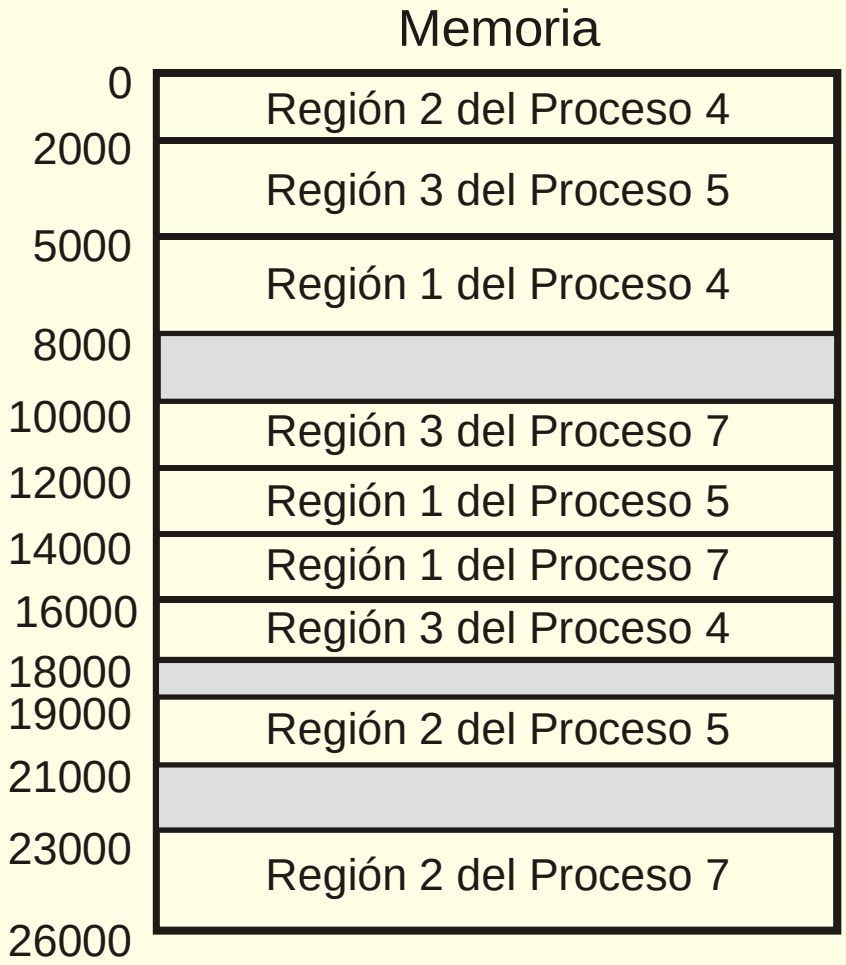

 

# Creación de espacio lógico independiente



# Segmentación



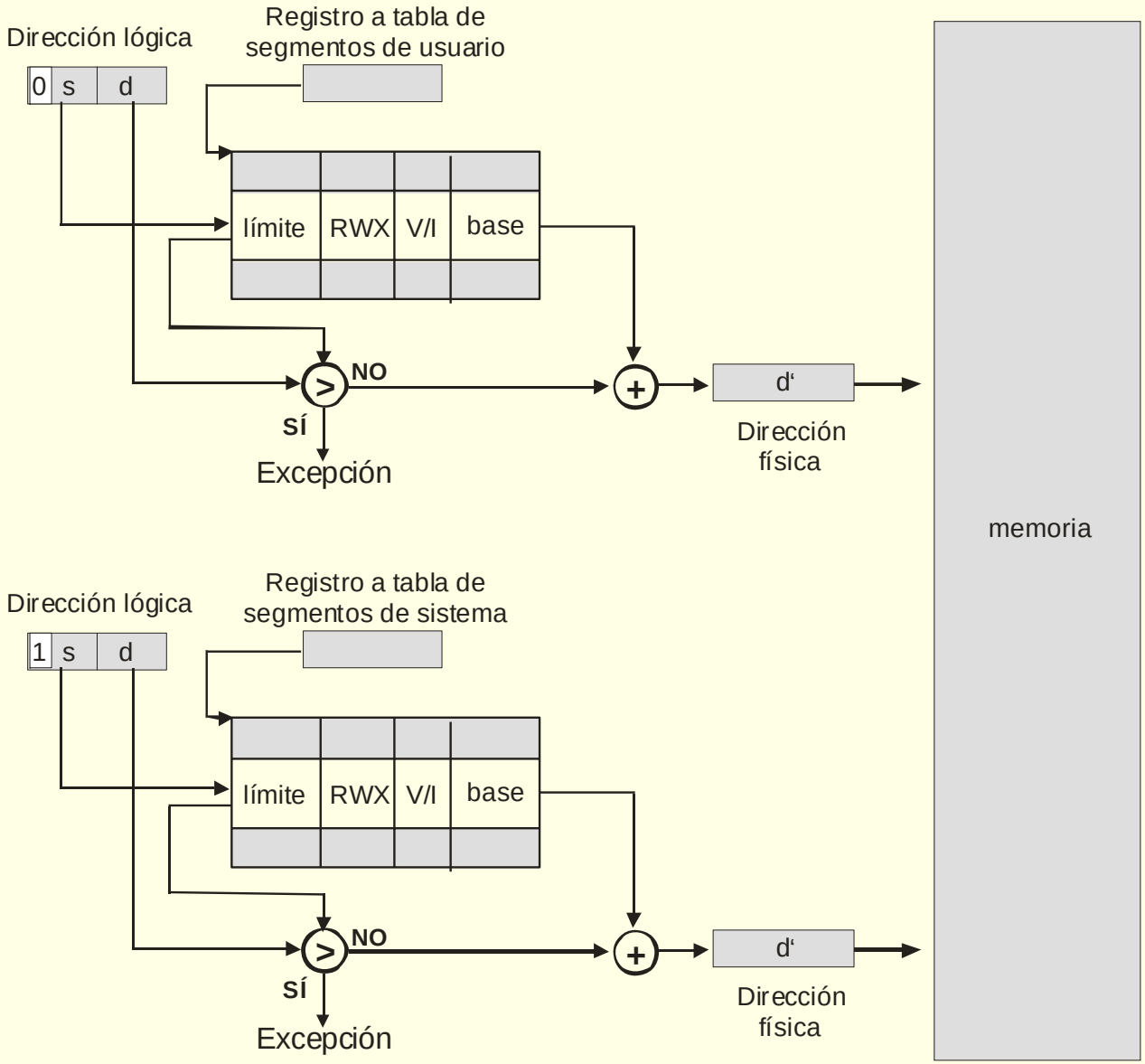

 

# Segmentación: Aspectos hardware

- Esquema que intenta dar soporte directo a las regiones
- Generalización de registro base y límite: 1 pareja/segmento
- Dirección lógica: núm. de segmento + dirección en segmento
- MMU usa una tabla de segmentos (TS)
- Entrada de TS contiene (entre otros):
  - r. base y límite del segmento
  - protección: RWX + bit de validez
- Dir. lógicas de usuario y de sistema (p.e. empiezan por 0 ó 1)

# Esquema de traducción con segmentación

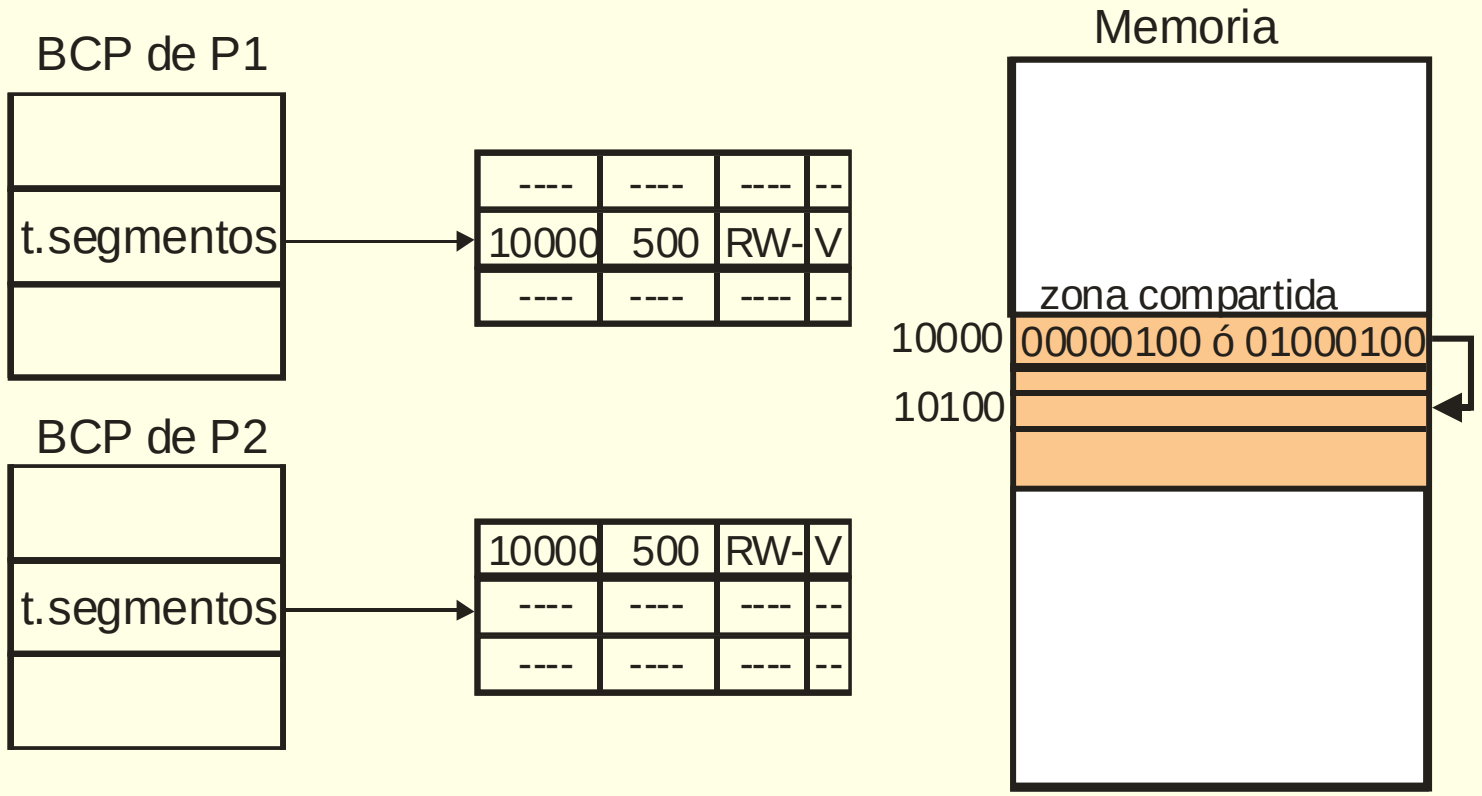


# Gestión de memoria basada en segmentación

- SO mantiene TSU por proceso (t. de regiones)
  - En c. contexto notifica a MMU qué TSU debe usar
- SO mantiene una única TSS que no cambia
  - Procesos comparten mapa del sistema operativo
  - SO interpreta directamente direcciones de usuario de p. actual
- Soporte directo de regiones (punto fuerte)
- Compartir: directo (región = segmento)
  - 2 entradas iguales en 2 TSU
  - Pero sigue habiendo problemas con autorreferencias
- Soporte de memoria virtual
  - No adecuado para m. virtual por tamaño variable de segmentos
- Nivel de procesos y regiones fusionado
  - Espacio de regiones se busca en m. del sistema no en mapa

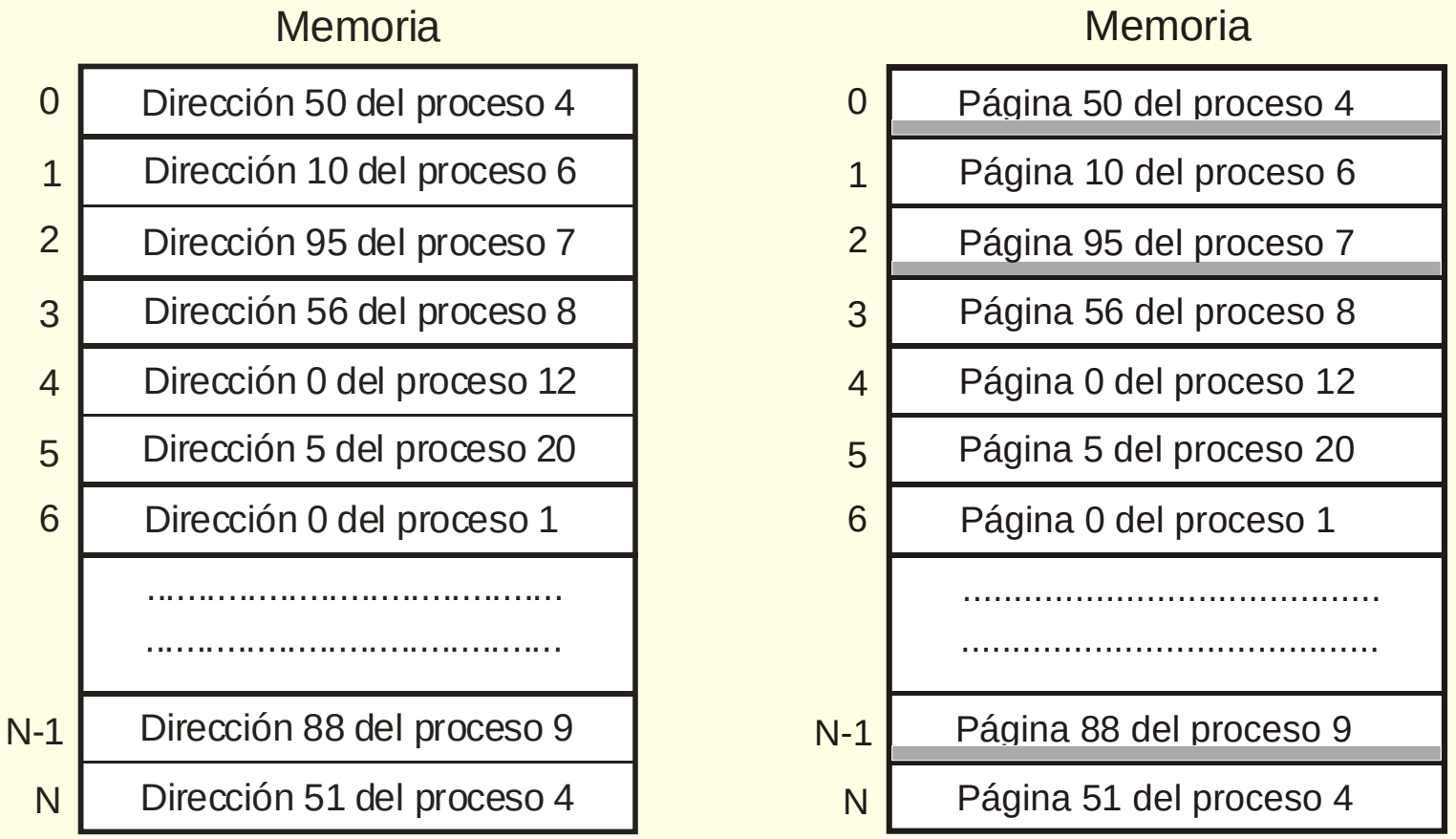
# Problemas al compartir una región



# Paginación: Fundamento

- Asignación contigua o segmentación:
  - Mal aprovechamiento por frag. externa
- Aprovechamiento óptimo es irrealizable
- Paginación: cambio de escala de byte a página
  - Cualquier página de proceso en cualquier marco de página
  - Peor aprovechamiento (f. interna) pero t. de traducción menor
  - Asignación no contigua: Reubicación no lineal
- Función de reubicación: tabla de páginas

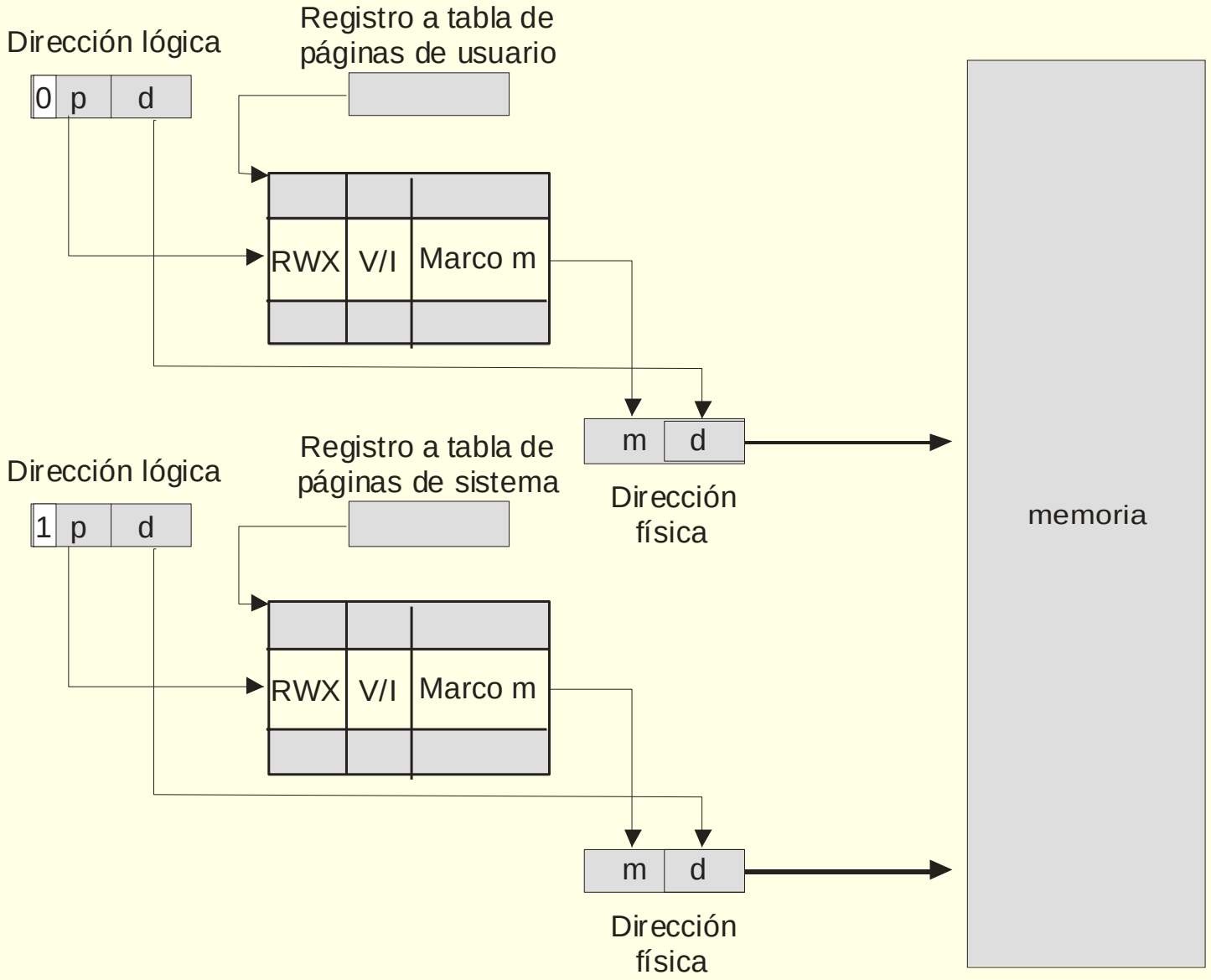
# Aprovechamiento óptimo vs con paginación



# Paginación. Aspectos hardware

- Mapa de memoria del proceso dividido en páginas
- Memoria principal dividida en marcos (tam. marco=tam. página)
- Tabla de páginas (TP): Asocia página y marco que la contiene
- Normalmente espacio lógico ≥ físico (bits de $p$ ≥ bits de $m$)
  - Excepción: Intel PAE (*Physical Address Extension*)
- Si MMIO: Para acceder a dispositivo crear entrada TP que asocie
  - dir. lógica → dir. de E/S
- Tabla de páginas única (bit $S$) vs. 2 tablas de páginas separadas
  - Usamos 2 tablas en los ejemplos por sencillez

# Esquema de traducción con TP usuario y sistema



# Contenido de entrada de TP

- Número de marco asociado
- Información de protección: RWX
- Bit de página válida/inválida
- Bit de página accedida (*Ref*)
- Bit de página modificada (*Mod*)
- Bit de desactivación de caché (para direcciones de E/S)
- Entrada de sistema (*S*): Presente en MMU con TP única
- Indicador de página global (*G*): accesible por todos
  - Presente en MMU con TP única
- Indicador de superpágina

# Tamaño de la página

- Condicionado por diversos factores contrapuestos:
  - Potencia de 2 y múltiplo de sector de disco
  - Compromiso (entre 1K y 16K)
  - Pequeño: Menor f. interna, mejor ajuste a conjunto de trabajo
  - Grande: Tablas más pequeñas, mejor rendimiento de disco
- Lo fija el procesador
  - Algunos permiten configurar distintos tamaños
  - Algunos implementan superpáginas
    - Por ejemplo: superpáginas de 4M y páginas de 4K

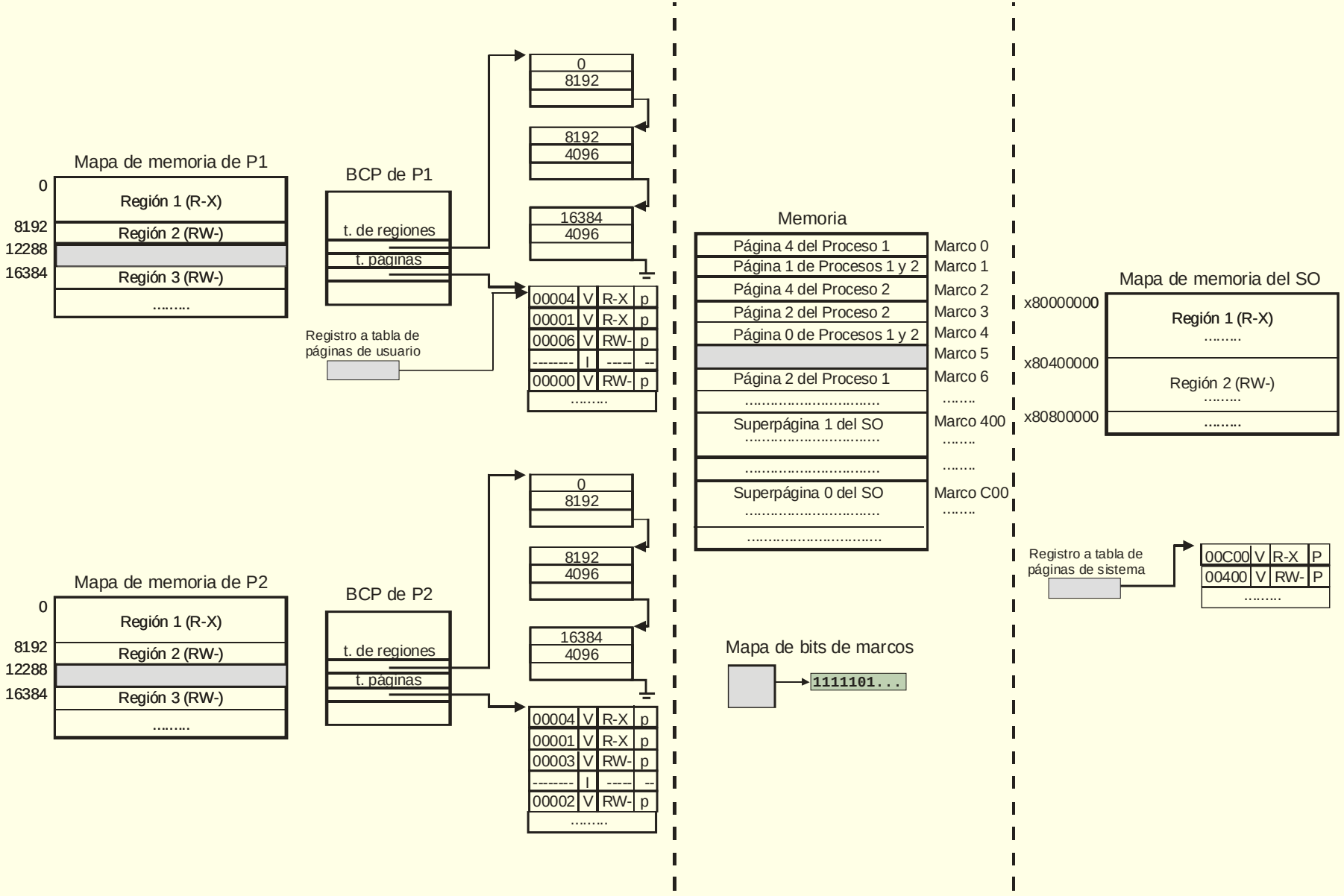
# Creación de espacio lógico independiente

- SO mantiene una TP por cada proceso
  - En c. contexto notifica a MMU cuál debe usar
- Con TP separadas, SO mantiene una única TP para propio SO
- Proceso modo sistema acceso directo a su mapa y al de SO
  - Procesos comparten mapa del sistema operativo
  - SO interpreta directamente direcciones de usuario de p. actual
  - SO usa asociaciones temporales para acceso a resto memoria

# Soporte de regiones, compartir y m. virtual

- No soporte directo de regiones pero fácil conseguirlo
  - Región ocupa número entero de páginas
  - SO mantiene tabla de regiones por cada proceso
  - Protección de región: uso de bits de protección de sus páginas
  - No reserva espacio para huecos: bit de validez desactivado
  - También soporte de regiones de SO (uso de superpáginas)
- Compartir región:
  - Entradas corresponden a mismo marco (contador de refs.)
- Permite esquemas de memoria virtual
  - Unidad de transferencia: página (tamaño fijo)
  - Uso de bit validez: página no residente con bit desactivado

# Visión global de la paginación



# Implementación de TP

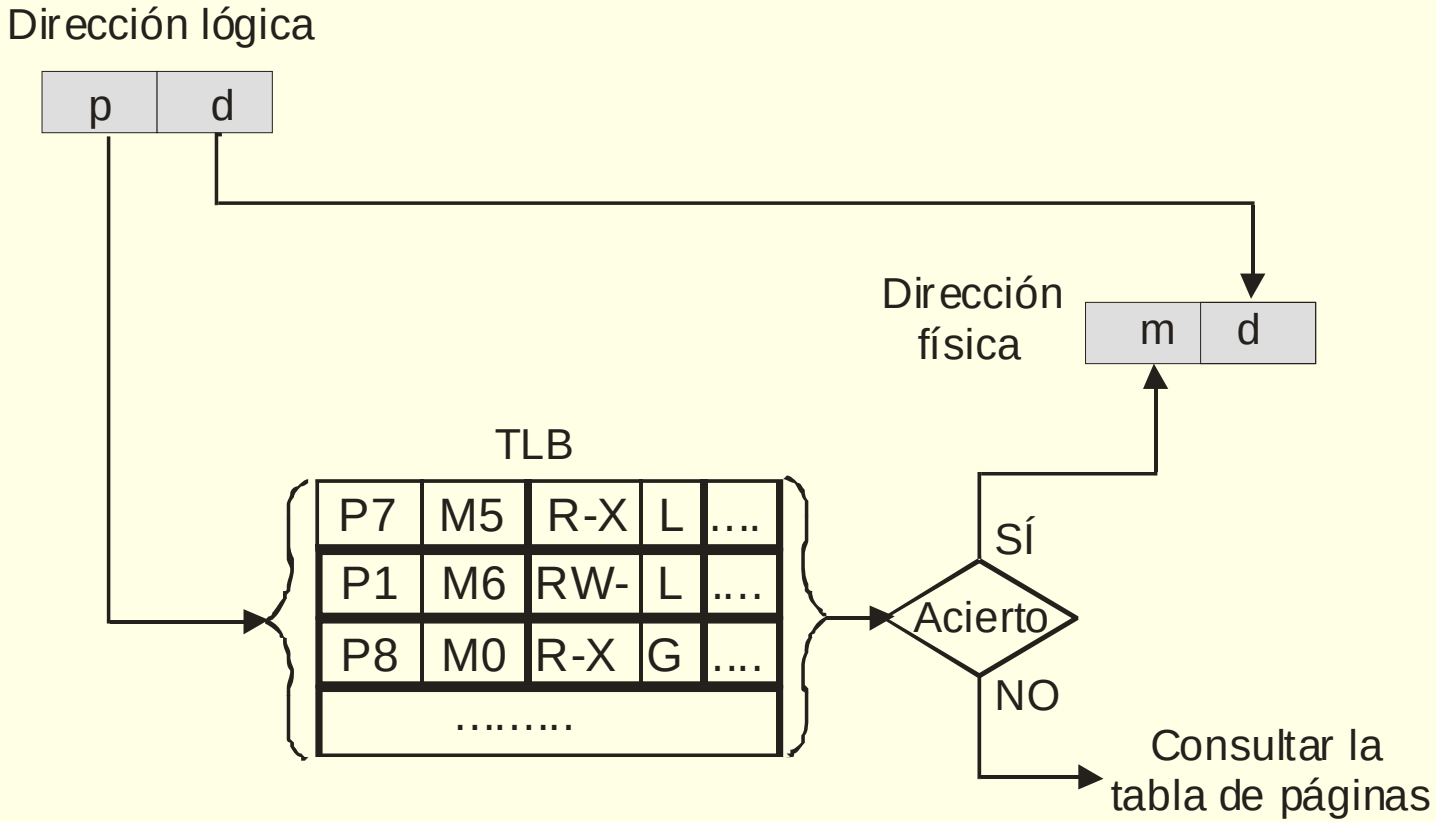
- TP se mantiene normalmente en memoria principal
- 2 problemas: eficiencia y gasto de almacenamiento
- Eficiencia:
  - Cada acceso lógico requiere 2 accesos a memoria principal
  - Solución: *caché* de traducciones → TLB
- Gasto de almacenamiento: Tablas muy grandes
  - Ejemplo: páginas 4K, dir. lógica 32 bits y 4 bytes/entrada
    - Tamaño TP: $2^{20}$ *4 = 4MB/proceso
  - Solución: tablas multinivel y tablas invertidas

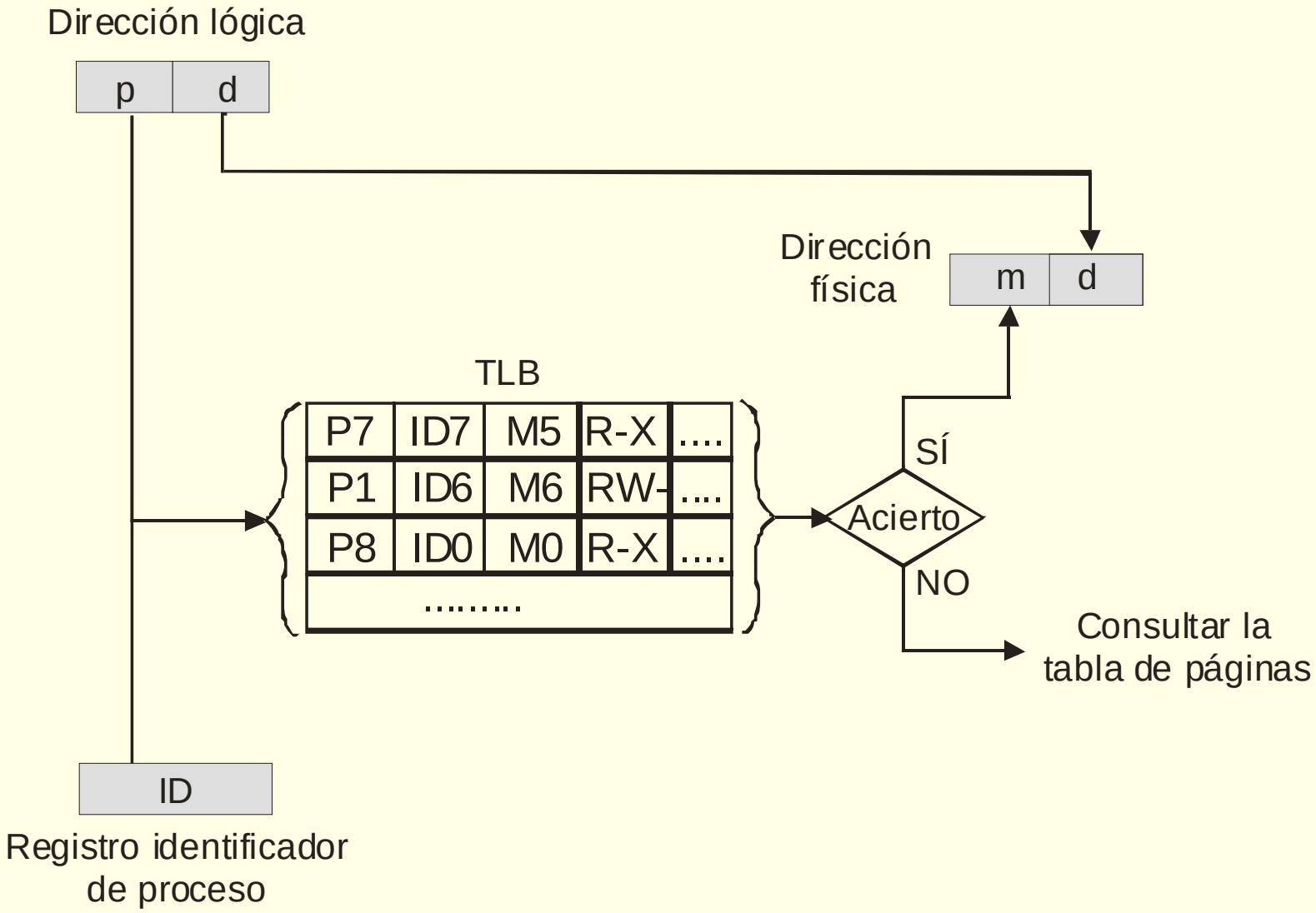
# *Translation Look-aside Buffer* (TLB)

- Memoria asociativa con info. sobre últimas páginas accedidas
- Entradas en TLB no incluyen información sobre proceso
  - **Invalida** TLB en c. contexto (no entradas de sistema)
- Entradas en TLB incluyen información sobre proceso
  - Registro UCP mantiene ID de p. actual:
    - No invalidar excepto si se reutilizan
- Gestionada por MMU: Si fallo usa la TP en memoria
  - MMU activa *Mod* y *Ref* en TP
- Pero no totalmente transparente al SO
  - Necesario invalidar entradas o todo su contenido

# TLB sin información de proceso



# TLB con información de proceso



# TLB gestionada por software

- Alternativa: traspasar al SO parte del trabajo de traducción
- MMU no usa tablas de páginas, sólo consulta TLB
- SO mantiene TPs que son independientes del HW
- Fallo en TLB → Activa SO
- SO se encarga de:
  - Buscar "a mano" en TP la traducción
  - Rellenar TLB con la traducción
  - Propagar bits *Ref* y *Mod* a TP
- Flexibilidad en diseño de SO pero menor eficiencia

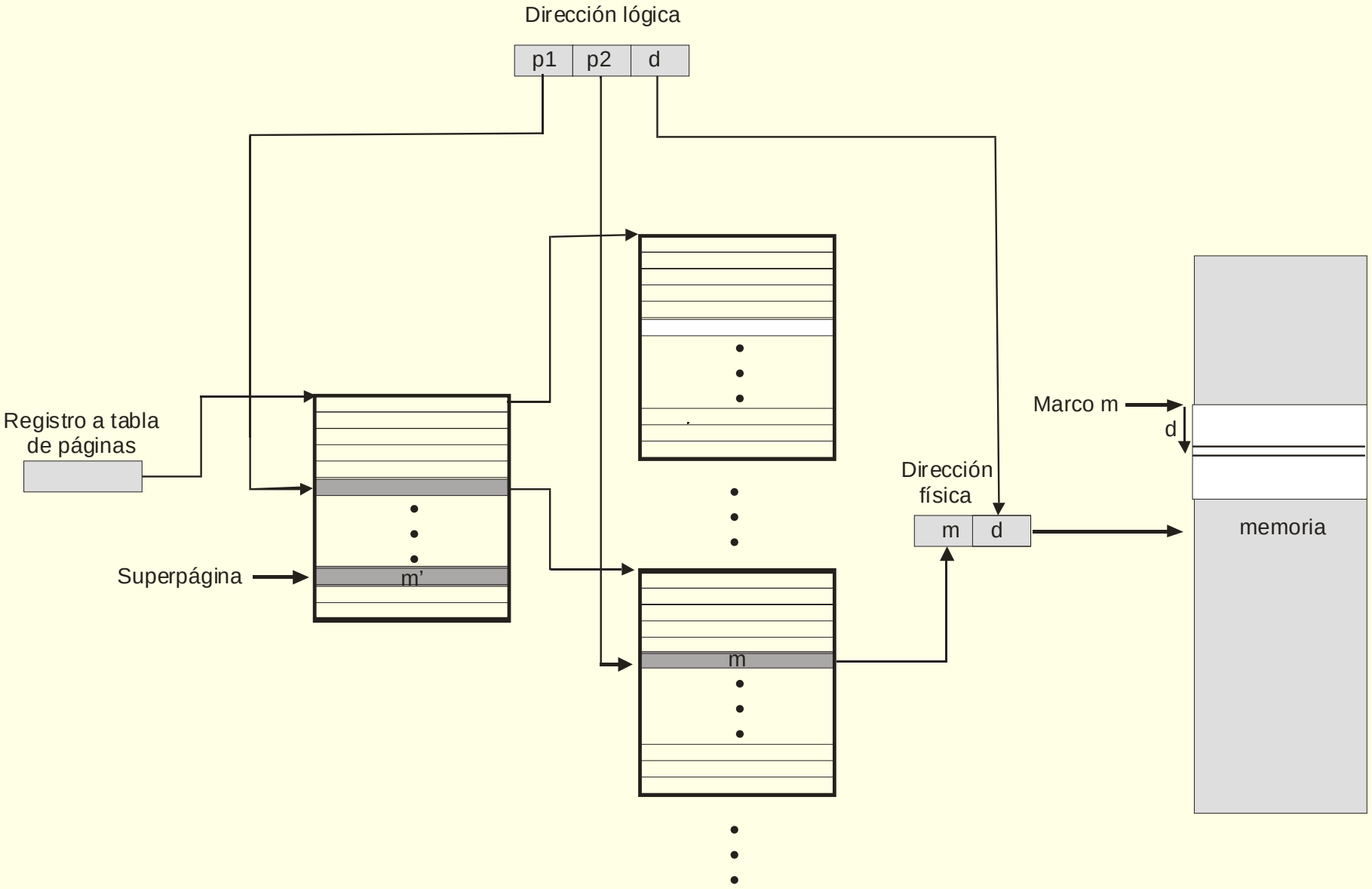
# La caché de memoria y la paginación

- ¿Requiere alguna gestión por parte del SO?
- Cachés físicas: accedidas con direcciones físicas
  - Acceso después de paginación: ineficiencia
    - Truco: *Virtual-Index/Physical-Tag*
      - índice de selección de caché usa campo *d* de direc. lógica
  - Son **transparentes** al S.O.
- Cachés virtuales: accedidas con direcciones físicas
  - Ventaja: Consulta TLB y caché en paralelo
  - Desventaja: homónimos y sinónimos (*virtual aliasing*)
  - Como TLB, pueden incluir información de proceso o no
  - Como TLB, **no transparentes** al SO
    - Necesario invalidar entradas o todo su contenido

# Tablas de páginas multinivel

- Fundamento
  - Tablas de páginas muy grandes con muchas entradas nulas
  - Fragmentar tabla y acceder mediante tabla maestra
- Tablas de páginas organizadas en M niveles:
  - Entrada de TP de nivel K apunta a TP de nivel K+1
  - Entrada de último nivel apunta a marco de página
- Dirección lógica especifica la entrada a usar en cada nivel:
  - 1 campo por nivel + desplazamiento
- Si todas las entradas de una TP son inválidas
  - No se almacena esa TP e inválida entrada de TP superior

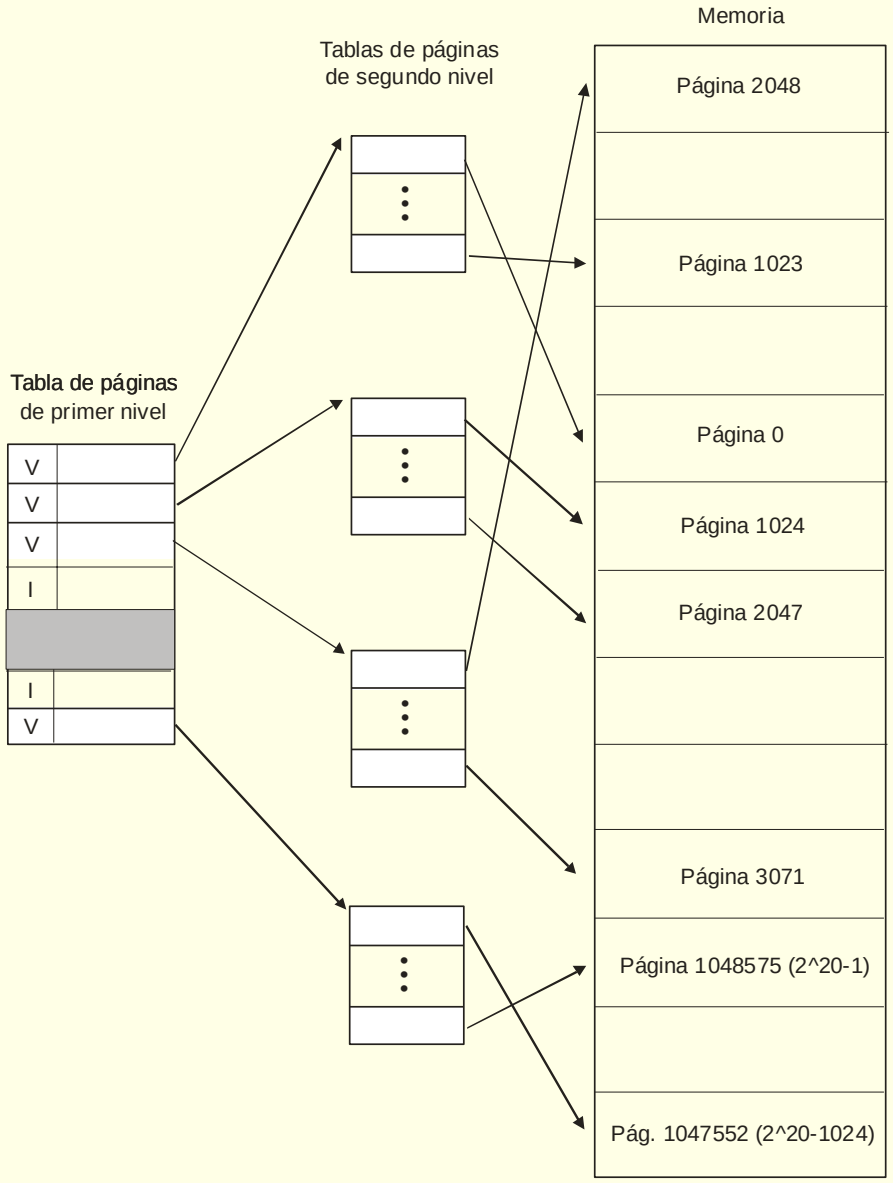
# Esquema de traducción con 2 niveles



# Ventajas de tablas de páginas multinivel

- Si proceso usa una parte pequeña de su espacio lógico
  - Ahorro en espacio para almacenar TPs
- Ejemplo: Proceso que usa 12MB superiores y 4MB inferiores
  - 2 niveles, pág. 4K, dir. lógica 32 bits (10 bits/nivel), 4B/entrada
    - Tamaño: 1 TP $N_1$ + 4 TP $N_2$= 5 * 4KB = 20KB (frente a 4MB)
- Ventajas adicionales:
  - Permite compartir TPs intermedias
  - Sólo se requiere que esté en memoria la TP de nivel superior
- Facilita implementación de superpáginas
  - Entrada de primer nivel de superpágina apunta a m. física

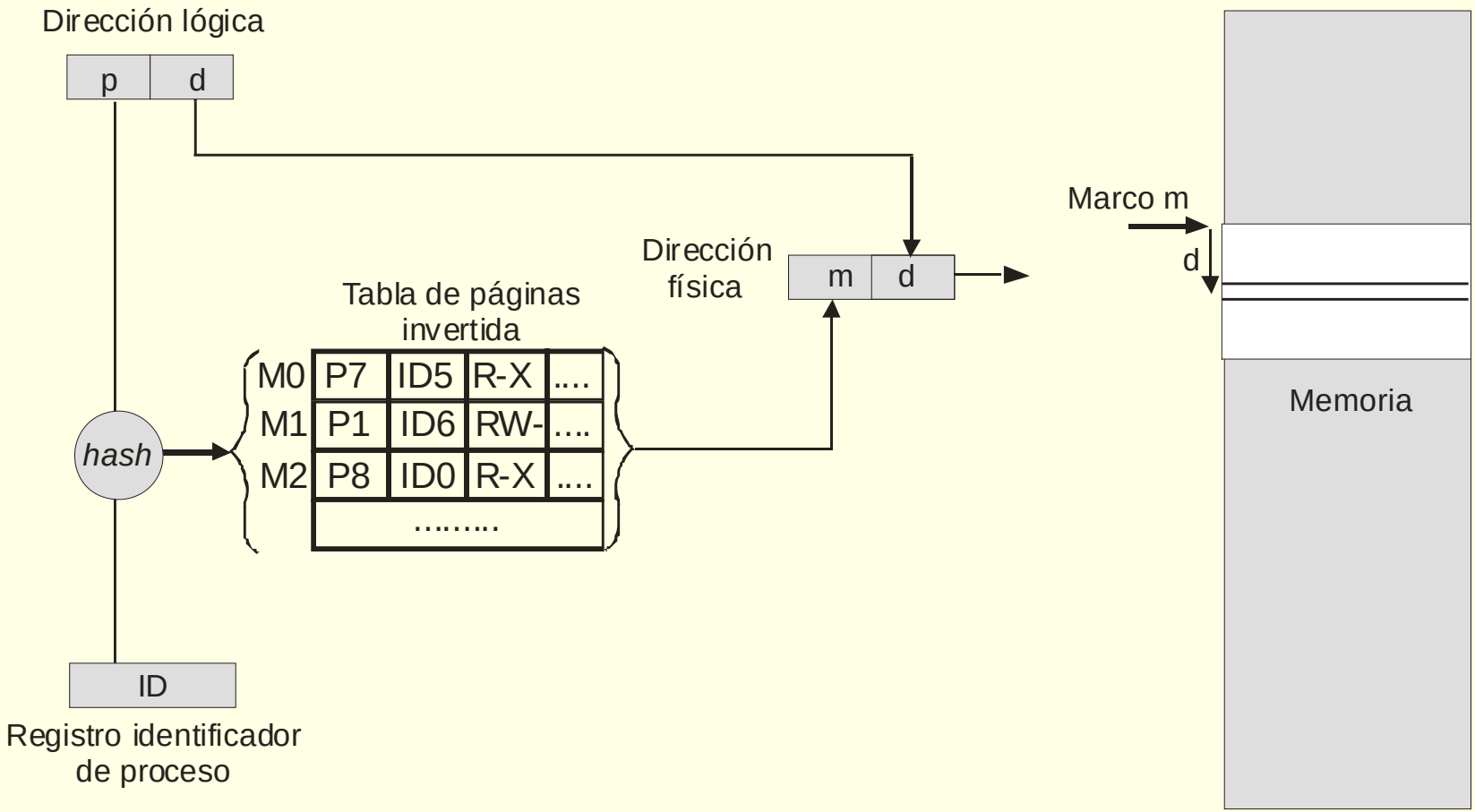
# Ventajas de tablas de páginas multinivel



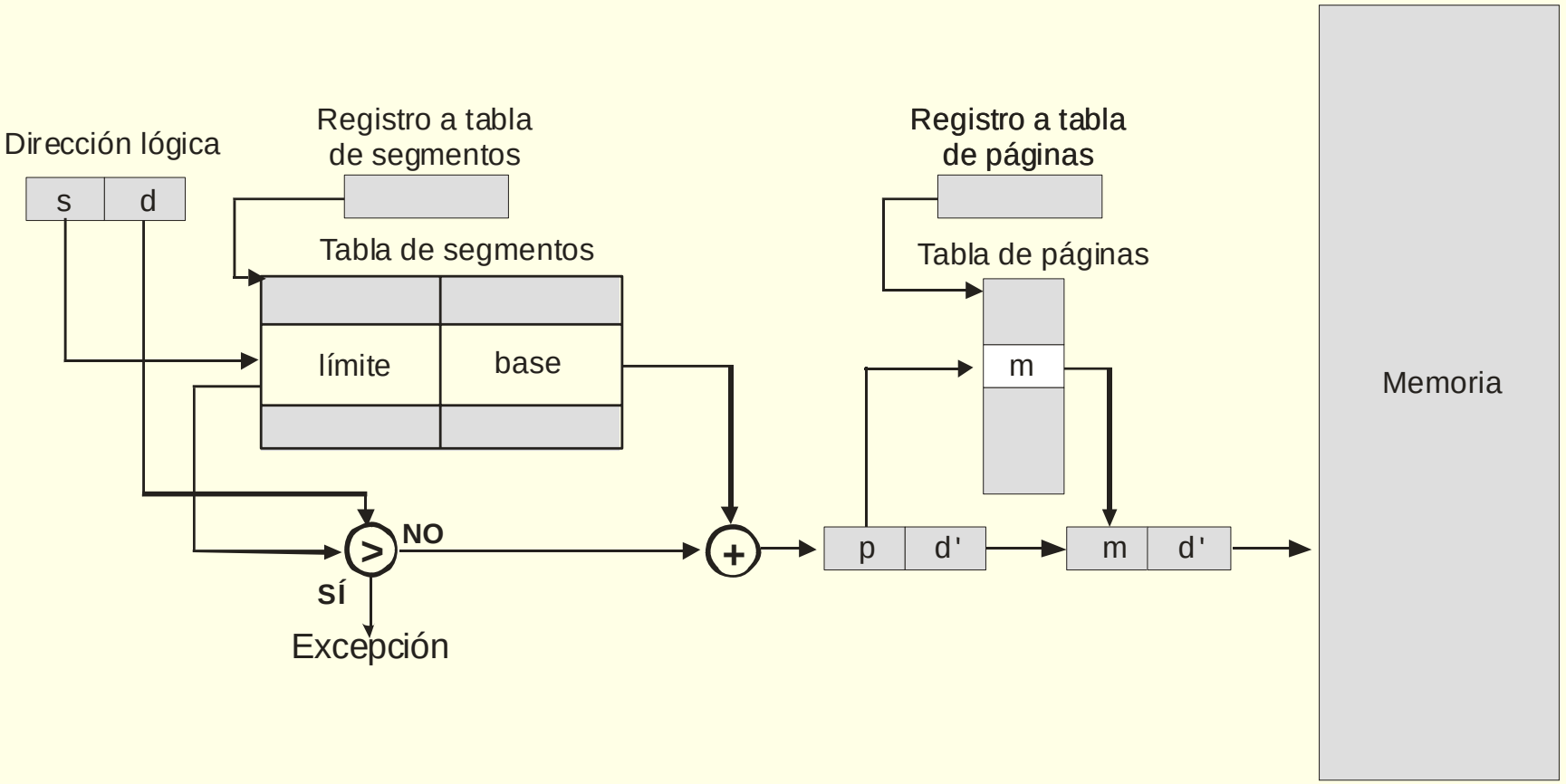
# Tabla de páginas invertida

- Procesadores actuales espacio lógico enorme (dirs. de 64 bits)
  - TPs muy grandes incluso usando multinivel
- Uso de TPs invertidas
  - Entrada por cada marco indica página almacenada en él
  - Necesario guardar núm. de página e id. de proceso
- Procedimiento de traducción:
  - MMU usa TLB convencional
  - Si fallo en TLB → se busca traducción en TP invertida
- Tabla *hash* para evitar búsqueda secuencial
- Difícil compartir. Alternativa: guardar marco en entrada de TP
- TP pequeña pero SO debe guardar info. de páginas no residentes

# Esquema de traducción con TP invertida



# Segmentación paginada: esquema de traducción

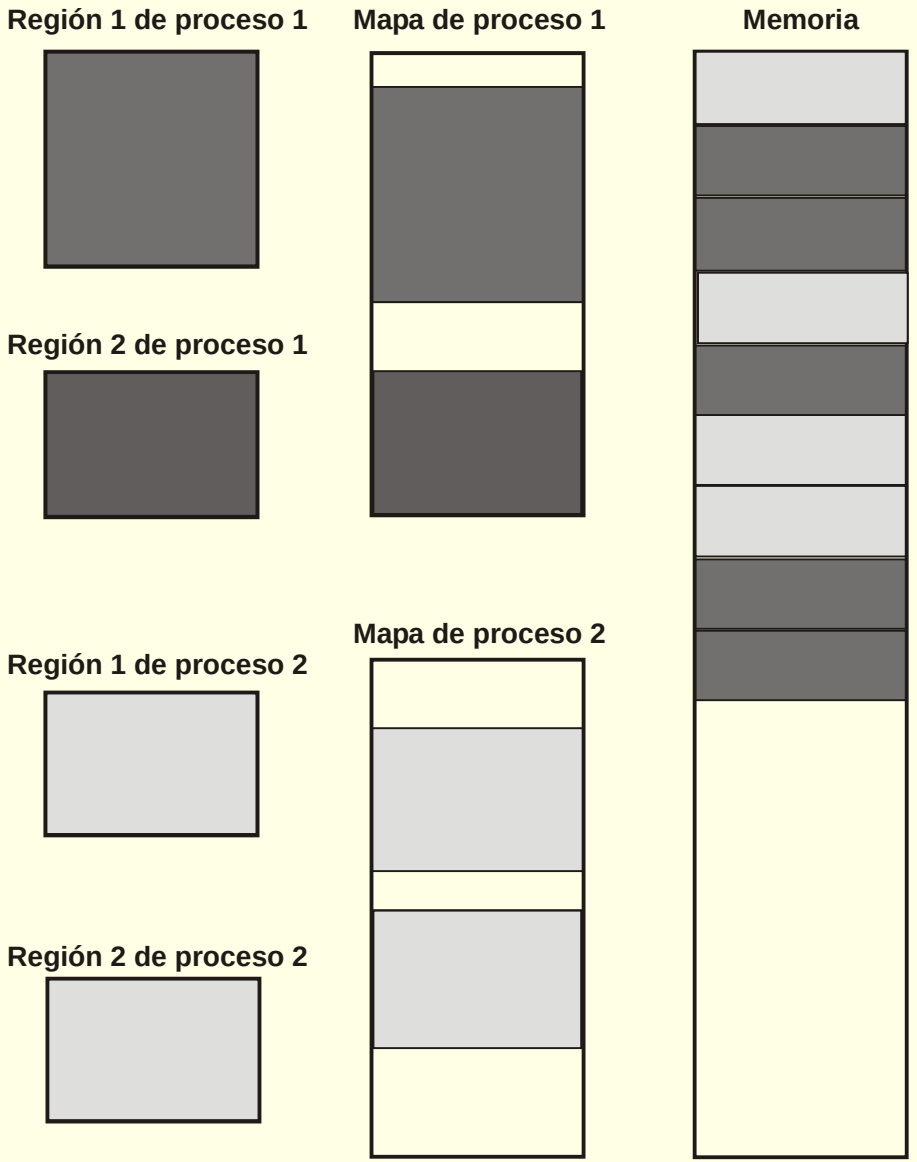



"Lo mejor de los dos mundos"

# Creación de espacios lógicos independientes

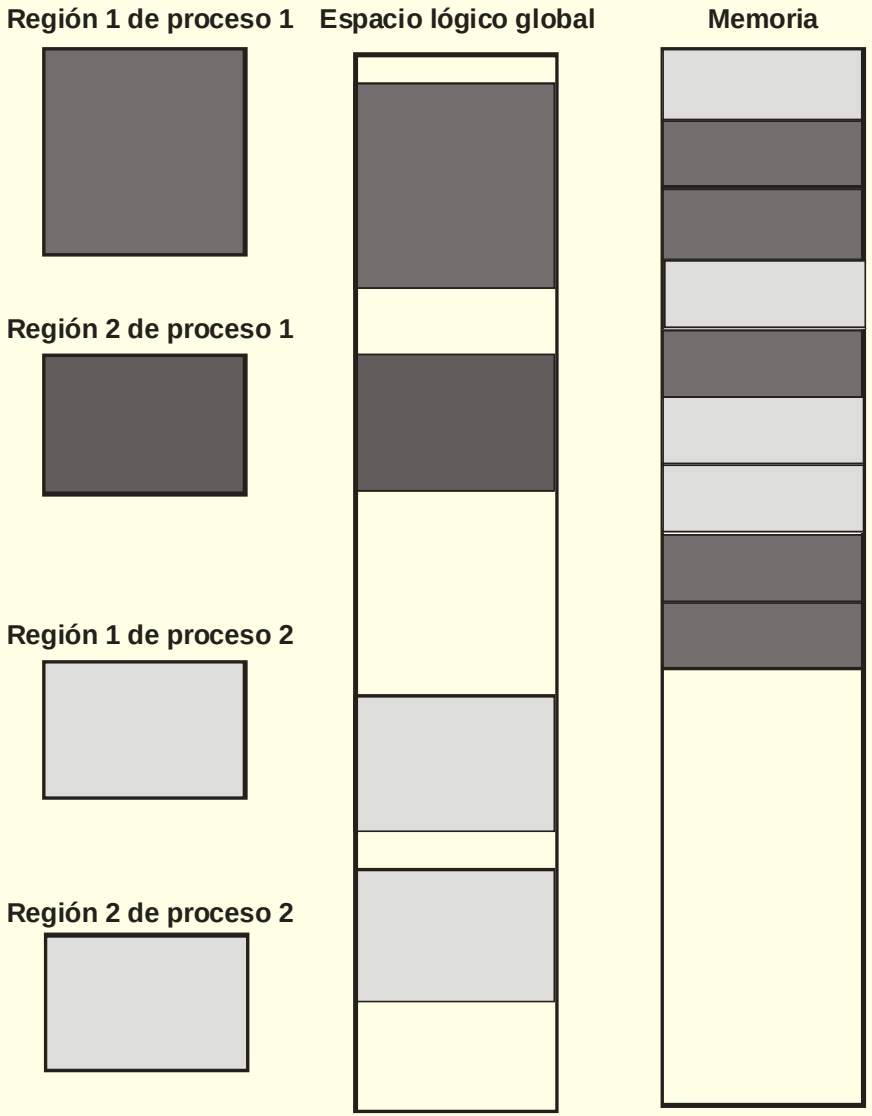
- 2 implementaciones alternativas por parte del SO
- Segmentación paginada local: 1TS/proceso y 1TP/proceso
  - Espacio lógico por proceso
  - Mayor similitud con paginación
- Segmentación paginada global: 1TS/proceso y TP única
  - Espacio lógico global
  - Mayor similitud con segmentación simple
  - Reduce problemas de coherencia (analizados al final del tema)
    - No requiere TLB con PID (ni caché virtual con PID)
    - No invalidación en cambio de contexto ni problema de sinónimos
- Mayoría de los SSOO (p.e. Linux) desactivan la segmentación
  - Política de mínimos requisitos facilita portabilidad
    - SO sólo requiere MMU con paginación

# Segmentación paginada local



# Segmentación paginada global



# SASOS, *Single Address-Space Operating Systems*

- Todos los procesos comparten espacio global único paginado
- Cada región tiene asignada dirección fija en espacio único
  - Región compartida vista por todos en mismo rango direcciones
  - Desaparece problema de autorreferencias
  - Paginación determina qué puede acceder cada proceso
  - Paginación proporciona reubicación de espacio lógico al físico
    - Buen aprovechamiento de memoria y soporte de memoria virtual
  - Reduce problemas de coherencia (analizados al final del tema)
    - No requiere TLB con PID (ni caché virtual con PID)
    - No invalidación en cambio de contexto ni problema de sinónimos

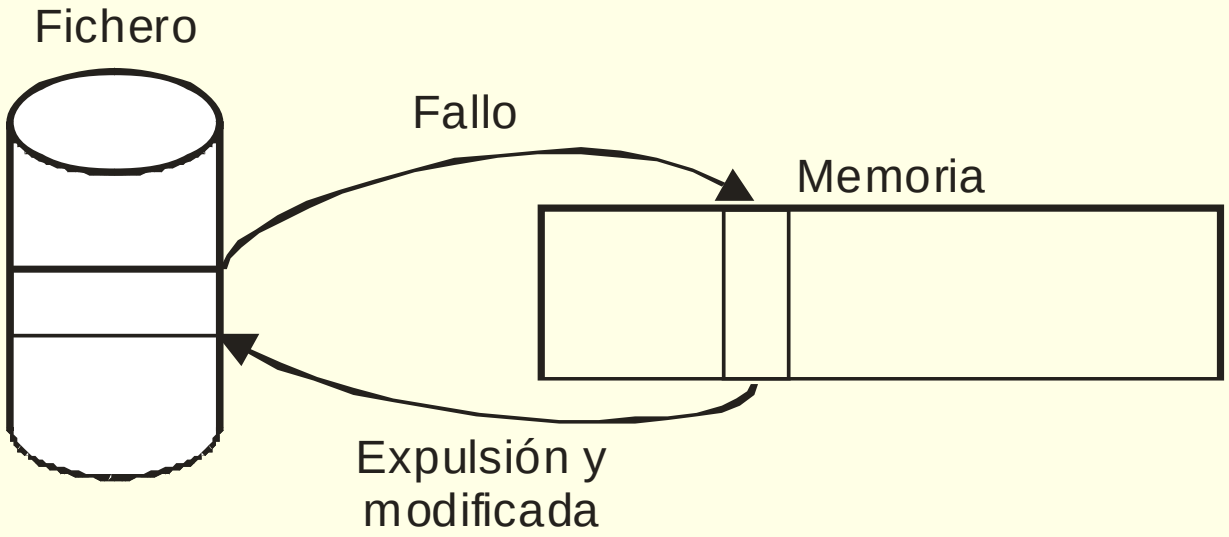
# M. virtual: Antecedentes

- **Intercambio** (*swapping*): más procesos de los que caben en mem
  - Disco (*swap*): respaldo de memoria
  - *Swap out*: expulsa/suspende proceso si no hay sitio
    - Diversos criterios para expulsar: mejor si bloqueado
  - *Swap in*: reanudación de proceso expulsado (y listo)
  - Preasignación de *swap* o no
- ***Overlays***: Programas más grandes que memoria disponible
  - No transparente a programador: info. de uso de módulos
  - Montador genera ejecutable con código de carga y descarga

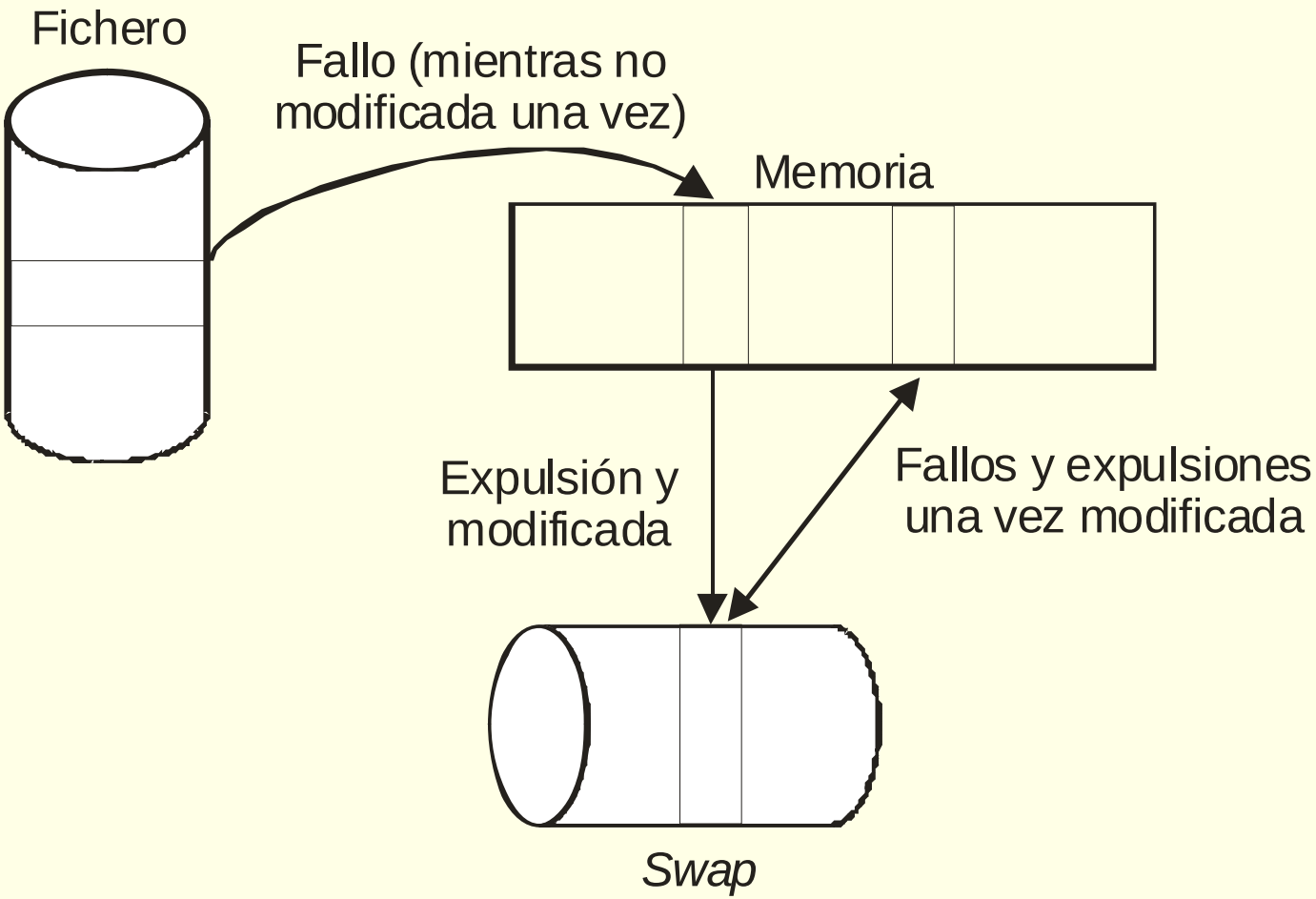
# Fundamento de la memoria virtual

- M. virtual: SO gestiona niveles de m. principal y m. secundaria
  - Sube por demanda; Baja por expulsión
- Aplicable por proximidad de referencias
  - Procesos sólo usan parte de su mapa en intervalo de tiempo
  - Parte usada (*cjto de trabajo*) en m. principal (*cjto residente*)
- Beneficios:
  - Aumenta el grado de multiprogramación
  - Permite ejecución de programas que no quepan en mem. ppal
- No adecuada para sistemas de tiempo real
- Basada en paginación: Uso del bit de validez
  - Página no residente se marca como no válida
  - En acceso: Excepción de fallo de página

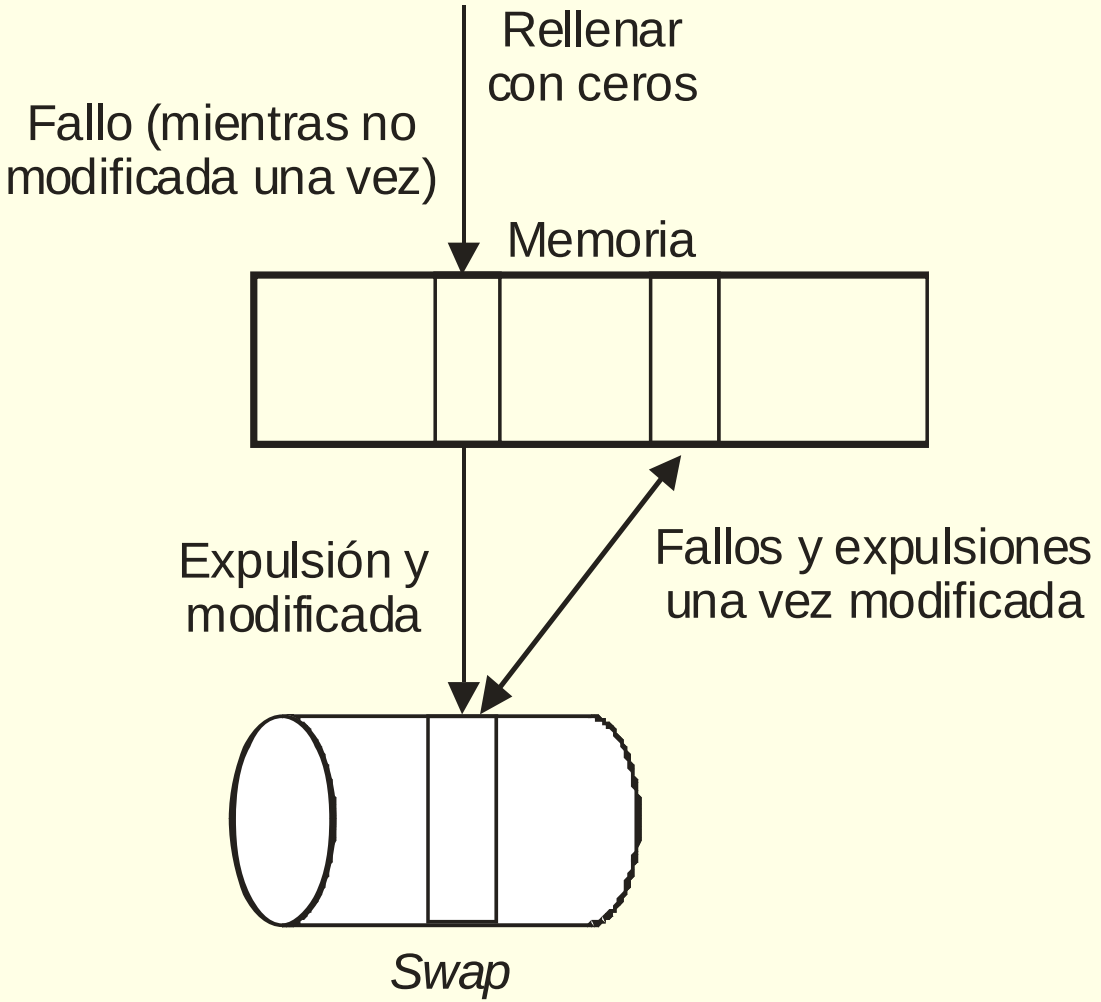
# Ciclo de vida de página compartida y en fichero



# Ciclo de vida de página privada y en fichero



# Ciclo de vida de página anónima



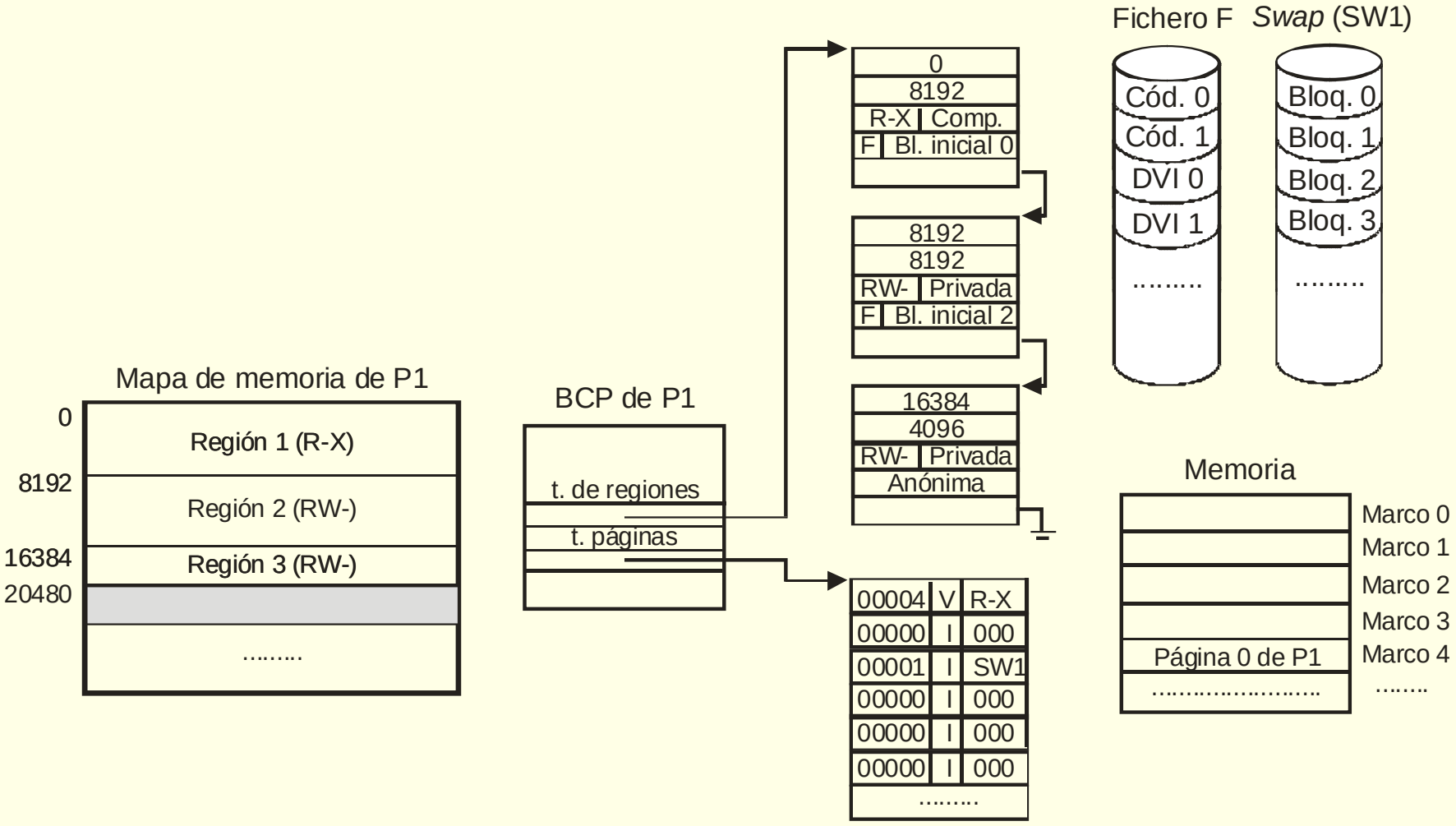

 

# Políticas de administración

- Localización
- Extracción
- Ubicación:
  - Cualquiera, aunque puede usarse coloración de páginas
- Reemplazo
- Actualización
  - Escritura diferida
- Reparto de espacio

# Política de localización: posibles ubicaciones

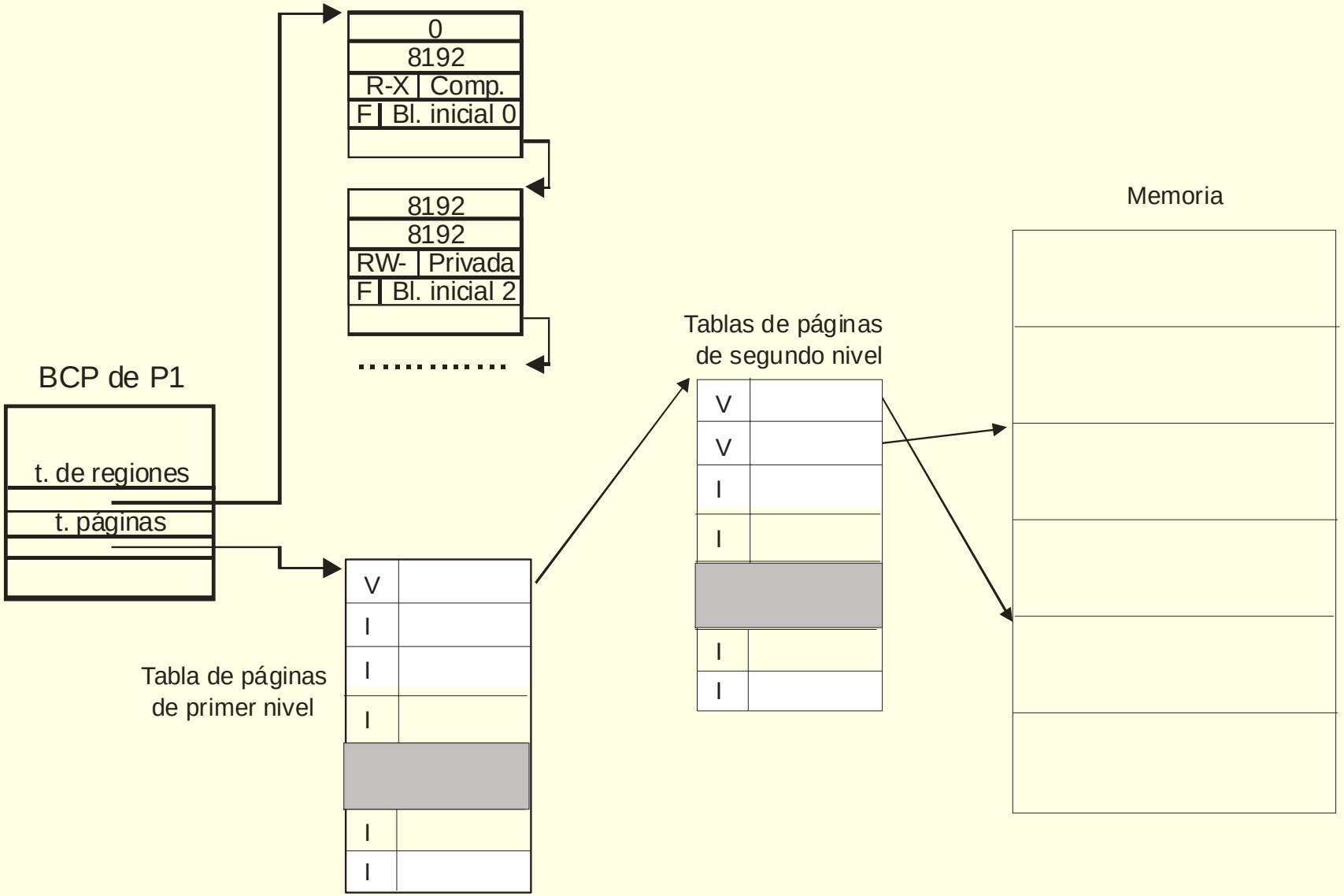


# Política de extracción: Fallo de página

- Si dirección inválida → Aborta proceso o le manda señal
- Si no hay ningún marco libre (consulta T. marcos)
  - Reemplazo: pág P marco M → P inválida
    - Si *mod* → **escribir** fichero (comp) o *swap* (priv)
- Hay marco libre (se ha liberado o lo había previamente):
  - Inicia **lectura** de página en marco M
  - Conecta entrada de TP a M
- Fallo de página en modo sistema no siempre es error:
  - Acceso a página de usuario no residente
  - Página de SO no residente o propagar cambios mapa SO
- Prepaginación: trae páginas por anticipado (no por demanda)

# Creación de tablas de páginas por demanda



# Política de reemplazo

- Tipo de reemplazo: local o global
- Factores a tener en cuenta:
  - Tiempo de residencia
  - Frecuencia de uso
  - *Frescura* de la página
- También en caché de sistemas de ficheros
- Objetivo: Minimizar la tasa de fallos de página.
  - Poca sobrecarga en tiempo y espacio y MMU estándar
- Algoritmo óptimo (MIN): Irrealizable
  - Página residente que tardará más en accederse

# Algoritmo FIFO

- Página que lleva más tiempo residente
- Fácil implementación:
  - Páginas residentes en orden FIFO
  - No requiere el bit de página accedida (*Ref*)
- Sobrecarga constante (independiente del tamaño de la memoria)
- No es una buena estrategia: basado sólo en tiempo de residencia
- Anomalía de Belady
  - Ejemplos donde: ↑ nº marcos ⇒ ↑ nº fallos

# Anomalía de Belady (wikipedia)

| Peticiones de página | 3 | 2 | 1 | 0 | 3 | 2 | 4 | 3 | 2 | 1 | 0 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Página más nueva | 3 | 2 | 1 | 0 | 3 | 2 | 4 | 4 | 4 | 1 | 0 | 0 |
| | | | | 3 | 2 | 1 | 0 | 3 | 2 | 2 | 2 | 4 | 1 | 1 |
| Página más antigua | | | | | 3 | 2 | 1 | 0 | 3 | 3 | 3 | 2 | 4 | 4 |

| Peticiones de página | 3 | 2 | 1 | 0 | 3 | 2 | 4 | 3 | 2 | 1 | 0 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Página más nueva | 3 | 2 | 1 | 0 | 0 | 0 | 4 | 3 | 2 | 1 | 0 | 4 |
| | | | | 3 | 2 | 1 | 1 | 1 | 0 | 4 | 3 | 2 | 1 | 0 |
| | | | | | 3 | 2 | 2 | 2 | 1 | 0 | 4 | 3 | 2 | 1 |
| Página más antigua | | | | | | 3 | 3 | 3 | 2 | 1 | 0 | 4 | 3 | 2 |

(rojo indica fallo de página)

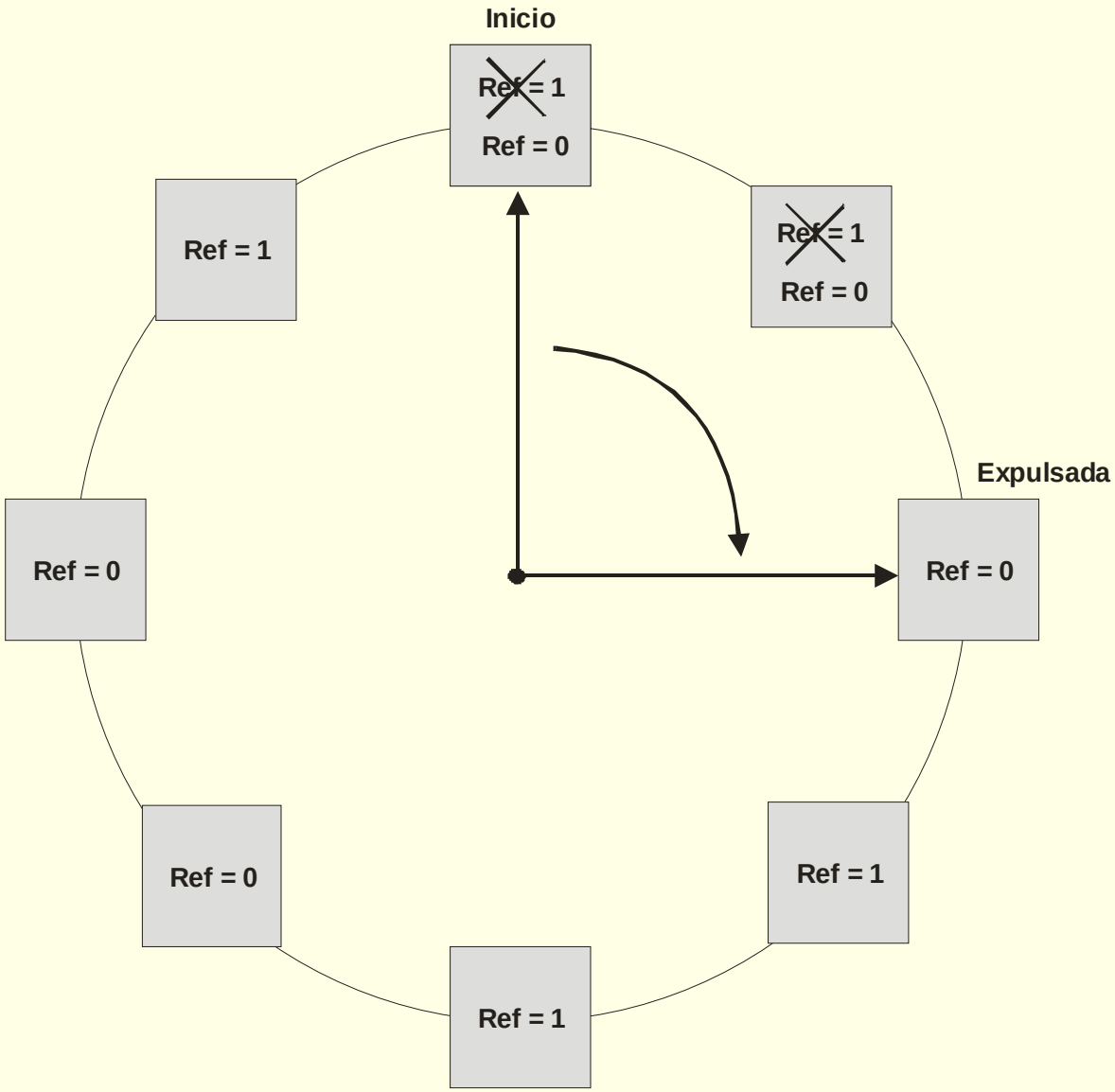
# Algoritmo LFU (*Least Frequently Used*)

- Página residente menos frecuentemente usada
- Problema: muchos accesos en una fase y ninguno después
- No anomalía de Belady: algoritmo de pila
- Difícil implementación estricta (hay aproximaciones):
  - Precisaría una MMU específica
- Se puede usar en caché de sistemas de ficheros
  - Sobrecarga en tiempo por cada acceso: log. tamaño de la caché

# Algoritmo LRU (*Least Recently Used*)

- Página residente menos recientemente usada (frescura)
- Proximidad de referencias: pasado reciente → futuro próximo
- No anomalía de Belady: algoritmo de pila
- Sutileza: ¿en tiempo de proceso o de sistema?
- Difícil implementación estricta (hay aproximaciones):
  - Precisaría una MMU específica
- Sí se usa como tal en caché de sistemas de ficheros
  - Sobrecarga en tiempo por cada acceso: constante
- No *scan-resistant*
  - Acceso a muchas páginas sólo una vez *poluciona* la memoria
  - Quedan residentes sólo esas páginas que no vuelven a usarse

# Algoritmo del reloj (o 2ª oportunidad)



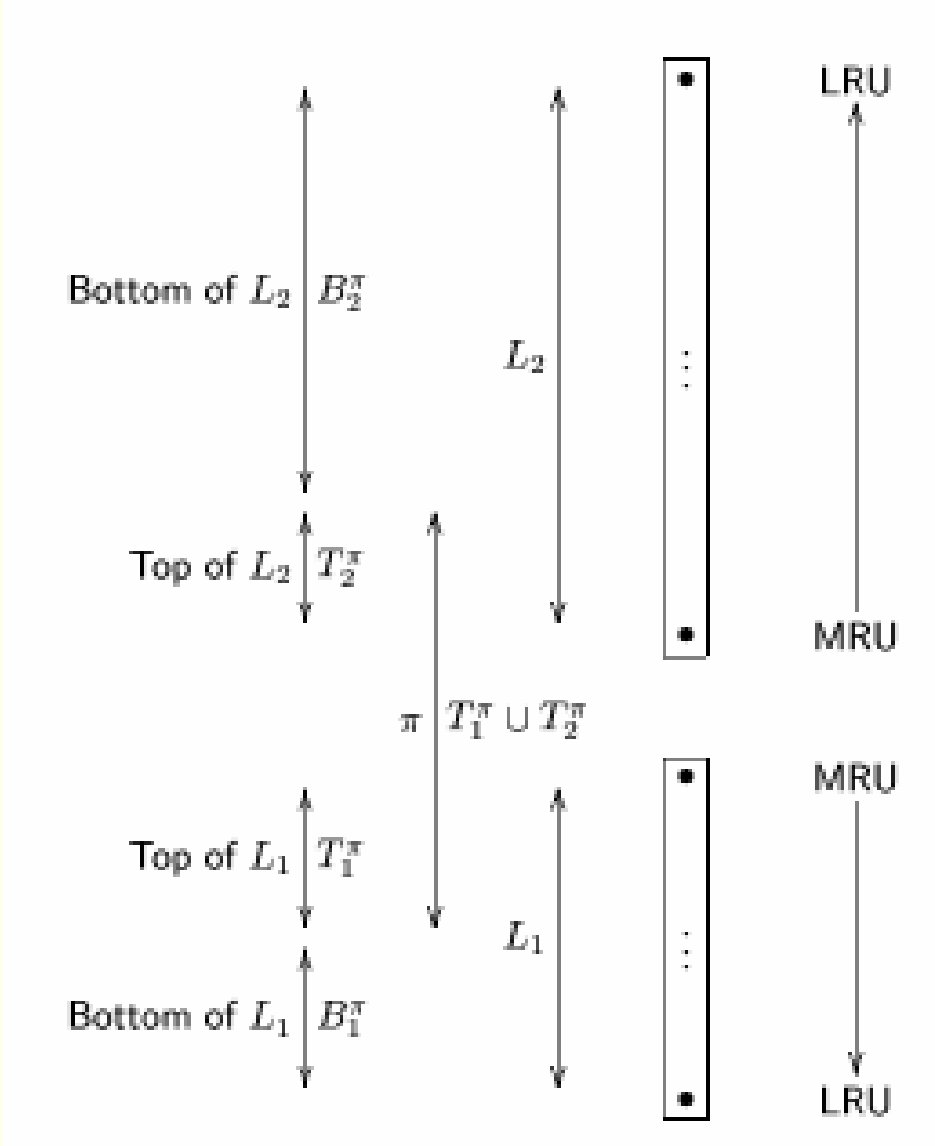
# Más allá de LRU: ARC

- ¿Cómo lograr ser *scan-resistant*?
  - Añadir frecuencia a frescura
  - LRU/2 (en gral., LRU/K) tiene en cuenta el penúltimo acceso
    - Sobrecarga en tiempo por cada acceso: log. tamaño de la caché
- ARC (*Adaptive Replacement Cache*) (Patente IBM)
  - 2 listas LRU: $L_1$ frescura, $L_2$ frecuencia
  - Guarda histórico (sobrecarga en espacio frente a LRU)
  - *Scan-resistant* y Autoconfigurable (gracias al histórico)
    - Si predomina frescura → reemplazo de $L_2$
    - Si predomina frecuencia → reemplazo de $L_1$
  - Precisaría una MMU específica
    - CAR (*Clock with Adaptive Replacement*): reloj + ARC
  - Se puede usar en caché de sistemas de ficheros
    - Sobrecarga en tiempo por cada acceso: constante

# ARC: Estructuras de datos

- Caché (DBL) gestiona información de $2c$ páginas:
  - Tiene copia de $c$ páginas (residentes) (ARC)
  - Guarda información histórica de otras $c$ páginas (no contenido)
  - Debe cumplirse: $|L_1| \leq c$ y $|L_1| + |L_2| \leq 2c$
- $L_1$ páginas accedidas sólo una vez dividida en:
  - $T_1$ páginas residentes y $B_1$ no residentes (histórico)
- $L_2$ páginas accedidas más de una vez dividida en:
  - $T_2$ páginas residentes y $B_2$ no residentes (histórico)
- Parámetro $p$: tamaño *deseado* para $L_1$
  - Fallo encuentra página en $B_1 \rightarrow$ incremento de $p$
  - Fallo encuentra página en $B_2 \rightarrow$ decremento de $p$

# ARC: Estructuras de datos



# ARC: algoritmo

ARC($c$)

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$.
INITIALIZATION: Set $p = 0$ and set the LRU lists $T_1$, $B_1$, $T_2$, and $B_2$ to empty.

For every $t \geq 1$ and any $x_t$, one and only one of the following four cases must occur.

Case I: $x_t$ is in $T_1$ or $T_2$. A cache hit has occurred in ARC($c$) and DBL($2c$).
Move $x_t$ to MRU position in $T_2$.

Case II: $x_t$ is in $B_1$. A cache miss (resp. hit) has occurred in ARC($c$) (resp. DBL($2c$)).

ADAPTATION: Update $p = \min\{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ |B_2|/|B_1| & \text{otherwise.} \end{cases}$

REPLACE($x_t, p$). Move $x_t$ from $B_1$ to the MRU position in $T_2$ (also fetch $x_t$ to the cache).

Case III: $x_t$ is in $B_2$. A cache miss (resp. hit) has occurred in ARC($c$) (resp. DBL($2c$)).

ADAPTATION: Update $p = \max\{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ |B_1|/|B_2| & \text{otherwise.} \end{cases}$

REPLACE($x_t, p$). Move $x_t$ from $B_2$ to the MRU position in $T_2$ (also fetch $x_t$ to the cache).

Case IV: $x_t$ is not in $T_1 \cup B_1 \cup T_2 \cup B_2$. A cache miss has occurred in ARC($c$) and DBL($2c$).

Case A: $L_1 = T_1 \cup B_1$ has exactly $c$ pages.
If ($|T_1| < c$)
Delete LRU page in $B_1$. REPLACE($x_t, p$).
else
Here $B_1$ is empty. Delete LRU page in $T_1$ (also remove it from the cache).
endif

Case B: $L_1 = T_1 \cup B_1$ has less than $c$ pages.
If ($|T_1| + |T_2| + |B_1| + |B_2| \geq c$)
Delete LRU page in $B_2$. if ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$).
REPLACE($x_t, p$).
endif

Finally, fetch $x_t$ to the cache and move it to MRU position in $T_1$.

Subroutine REPLACE($x_t, p$)
If ( ($|T_1|$ is not empty) and ( ($|T_1|$ exceeds the target $p$) or ($x_t$ is in $B_2$ and $|T_1| = p$)) )
Delete the LRU page in $T_1$ (also remove it from the cache), and move it to MRU position in $B_1$.
else
Delete the LRU page in $T_2$ (also remove it from the cache), and move it to MRU position in $B_2$.
endif

# *Buffering* de páginas

- Reemplazo bajo demanda: Mejor por anticipado
- Reserva de marcos libres
- Fallo de página: siempre usa marco libre (no reemplazo)
- Si nº marcos libres < umbral
  - "demonio de paginación" aplica algoritmo de reemplazo
    - páginas no modificadas → lista de marcos libres
    - páginas modificadas → lista de marcos modificados
      - cuando se escriban a disco pasan a lista de libres
- Si se referencia una página mientras está en estas listas:
  - fallo de página la recupera directamente de la lista (no E/S)

# Caché de páginas

- Encontrar eficientemente si página está residente:
  - Necesario en *buffering* y al compartir
  - [Fichero|Disp. *swap*, nº bloque] → {nº marco | !residente}
  - Págs. anónimas no en caché de páginas
- Cargar página de fichero o *swap* en fallo
  - Insertar en caché de páginas
- Sistema de ficheros también incluye caché de bloques
  - Tendencia: fusión de caché de páginas y de bloques

# Retención de páginas en memoria

- Páginas marcadas como no reemplazables
- Se aplica a páginas del propio SO
  - SO con páginas fijas en memoria es más sencillo
- También se aplica mientras se hace DMA sobre una página
- Servicio para fijar en memoria una o más páginas de su mapa
  - Adecuado para procesos de tiempo real
  - Puede afectar al rendimiento del sistema
  - En POSIX servicio `mlock`

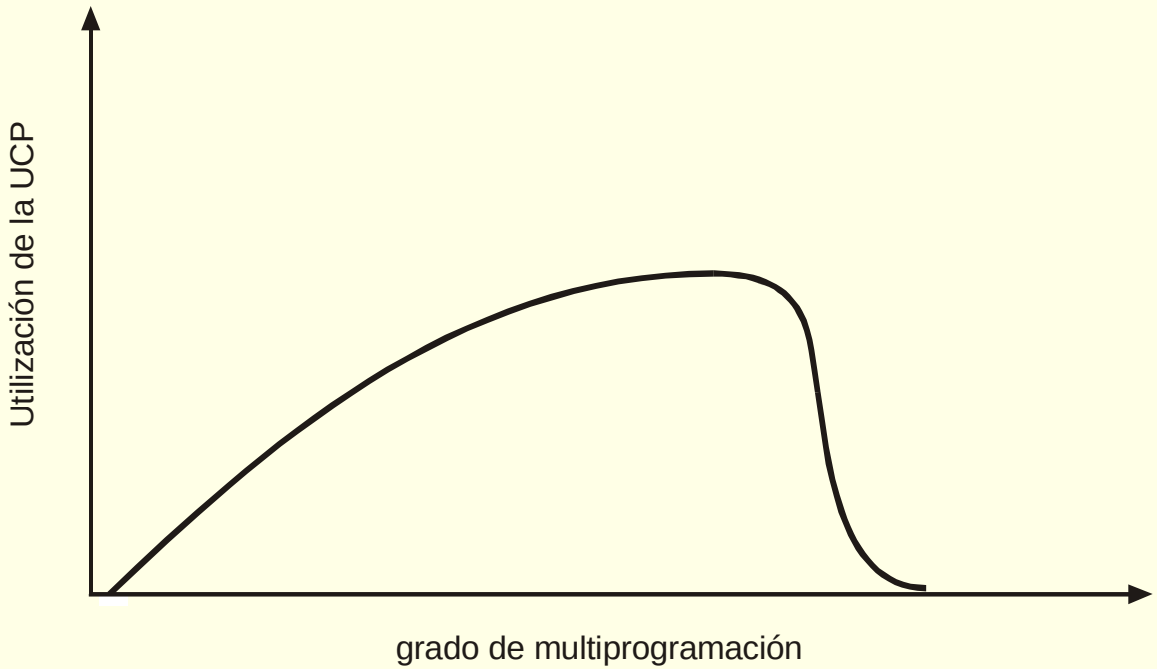
# Política de reparto de espacio

- Estrategia de asignación fija (reemplazo local)
  - Nª marcos asignados a proceso (cjto residente) es constante
  - No se adapta a las distintas fases del programa
  - Comportamiento relativamente predecible
  - Arquitectura impone nº mínimo
- Estrategia de asignación dinámica
  - Nº marcos varía según evolución de proceso(s)
  - Asignación dinámica + reemplazo local
    - comportamiento relativamente predecible
  - Asignación dinámica + reemplazo global
    - comportamiento difícilmente predecible

# Hiperpaginación (*Thrashing*)

- Tasa excesiva de fallos de página de proceso o de sistema
- Con asignación fija: Hiperpaginación en proceso
- Con asignación variable: Hiperpaginación en el sistema

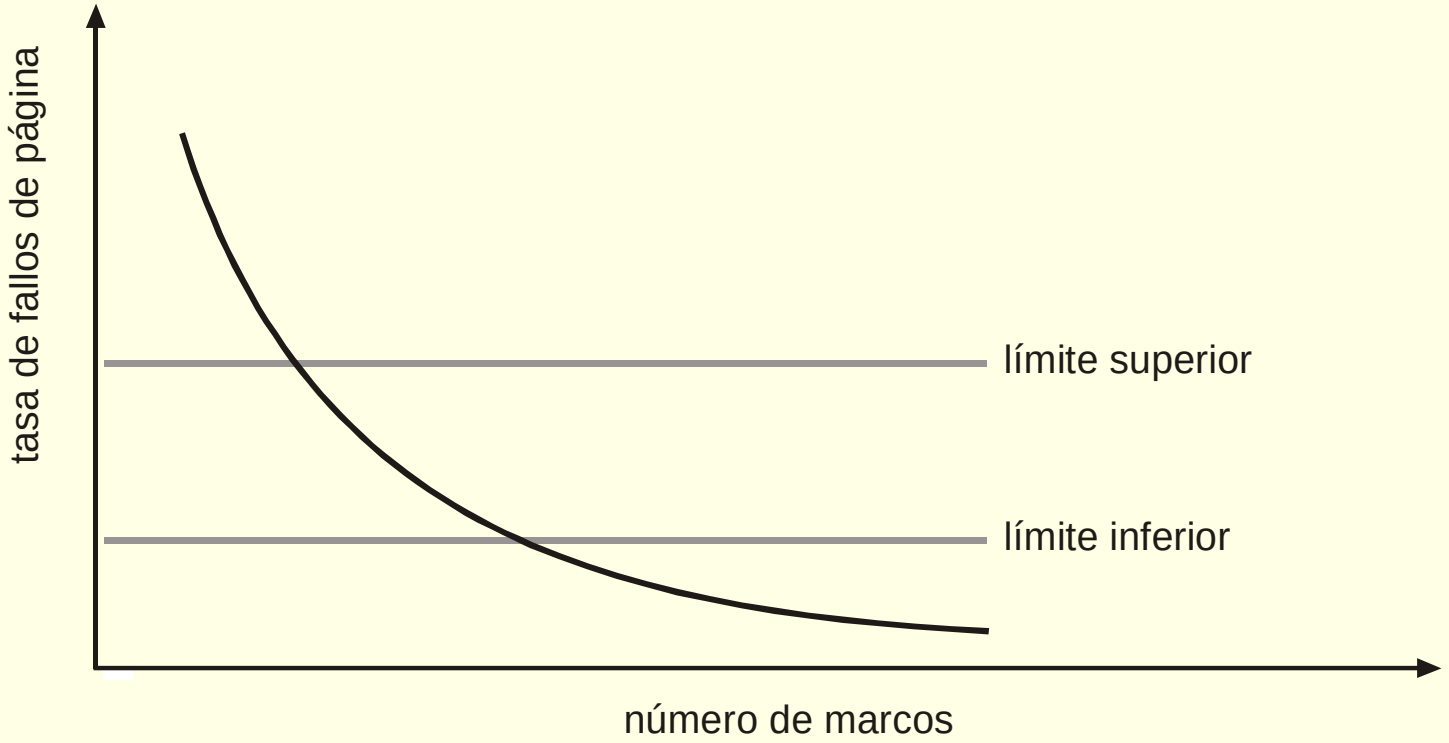


# Políticas de control de carga

- Estrategia del conjunto de trabajo
  - Páginas usadas por proceso en últimas N referencias
  - Si conjunto de trabajo decrece se liberan marcos
  - Si conjunto de trabajo crece se asignan nuevos marcos
    - si no hay disponibles: suspender proceso(s)
  - Requiere MMU específica
- Estrategia basada en frecuencia de fallos de página (PFF)
- Control de carga y reemplazo global
  - No control de hiperpaginación
  - Algoritmo de control de carga empírico
    - Si nº marcos frecuentemente debajo de umbral

# Estrategia basada en frecuencia de fallos

- Si tasa < inferior se liberan marcos aplicando reemplazo
- Si tasa > límite superior se asignan nuevos marcos
  - Si no marcos libres se suspende algún proceso



# Dispositivo de paginación (*swap*)

- Disco, partición o fichero (más lento)
  - Incorporación dinámica y configurable de espacio de *swap*
- Estructura: cabecera y bloques; mapa de bits sólo en memoria
- Preasignación de espacio
  - Al crear región privada o sin soporte
  - Permite detectar síncronamente la falta de espacio de *swap*
- Sin preasignación de espacio
  - En primera expulsión se le asigna espacio en *swap*
  - Mejor aprovechamiento de espacio de almacenamiento
- Regiones privadas o sin soporte usan el *swap*
  - Compartidas con soporte usan directamente fichero
- Bloque de *swap* puede compartirse: contador de referencias

# Compartimiento de páginas

- Escenarios:
  - Zona de memoria compartida
  - Compartir código de programa o biblioteca
  - Fichero proyectado en modo compartido
  - Regiones compartidas después de `fork`
- Compartir igual que con paginación convencional pero:
  - Caché de páginas para localizar pág compartida ya residente
  - Traducción inversa al expulsar página compartida
    - De marco de página a entradas de TP que lo referencian

# Duplicado perezoso de páginas

- Escenarios de uso duplicado:
  - Datos con valor inicial de programas y bibliotecas
  - Regiones privadas después de `fork`
  - Fichero proyectado en modo privado
- Optimización: Duplicado por demanda (*copy-on-write*, COW)
  - Se comparte una página mientras no se modifique
  - Si un proceso la modifica se crea una copia para él
- Implementación de COW
  - Se comparten páginas de regiones duplicadas pero:
    - se marcan de sólo lectura en TP (no en t. regiones)

# Tratamiento del fallo de COW

- Si dirección $\notin$ región WR $\rightarrow$ Aborta (o señal) proceso
- Si ref > 1
  - Reserva marco libre, copia contenido y conecta a TP
  - Devuelve permiso WR a entrada, ref--, no inserta caché págs
- Si ref == 1
  - Devuelve permiso WR a entrada, elimina de caché págs
- Si página con bloque de *swap* asignado, desvincular del mismo

- **Recordatorio importante**:
  - Sólo se asigna espacio (marcos) en fallo de página y de COW
    - **Nunca** al crear el mapa o una región del mismo

# Gestión de memoria del SO

- Gestión de la memoria física
  - En iniciación SO recibe información de memoria física
  - SO crea e inicia tabla de marcos
- Gestión del mapa del SO
  - Fase inicial: SO crea e inicia TP del sistema
    - SO contiguo en memoria → uso de superpáginas
  - Fase estable: Marcos restantes para procesos y SO
    - Para SO: asociación de marco con mapa SO
    - Para proceso: asociación con mapa usuario + a. temporal de SO
    - Alternativa: mapa de SO incluye toda memoria física
      - ► No requiere asociación temporal pero menos flexible

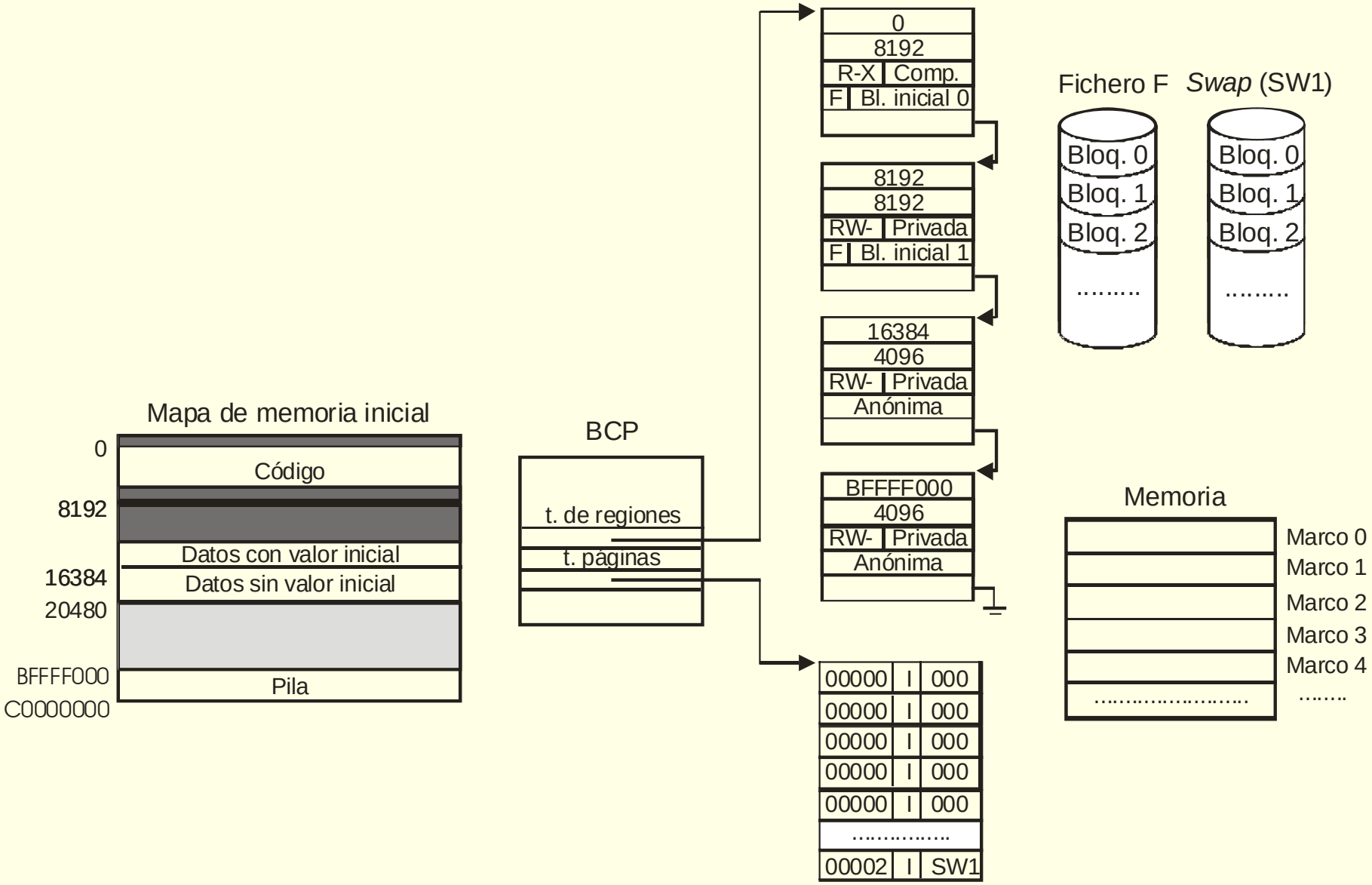
# Esquemas de asignación de espacio del SO

- Reserva de espacio contiguo lógico versus físico
- Distintos gestores de memoria del SO:
  - Reserva de págs contiguas en espacio físico (get_free_pages)
  - Reserva de un cierto tamaño (kmalloc)
  - Reserva de objetos del mismo tipo (p.ej. BCP) (*slabs*)
  - Reserva de espacio contiguo lógico (vmalloc)
- Diversos gestores usan cachés
  - Equilibrio entre reclamar espacio de caché o de usuario
- SO con mapa residente vs con parte del mapa paginable
- En multiprocesadores, prerreservas por cada UCP

# Operaciones en el nivel de regiones

- Crear una región: No asigna m. principal ni entradas TP
  - Busca zona libre en mapa de proceso usando t. regiones
    - Excepto si ubicación de la región está prefijada
  - Reserva y rellena entrada t. regiones
    - Si preasignación, reserva *swap*
- Eliminar una región:
  - Libera entrada t. regiones e invalida entradas TP
  - Marcos y bloques de *swap*: ref--
- Redimensionar una región *(heap* o pila)
  - Ajusta entrada t. regiones
    - Decrece: Invalida entradas TP + marcos y *swap*: ref--
    - Crece: Nada más, excepto reserva *swap* si preasignación

# Creación del mapa inicial

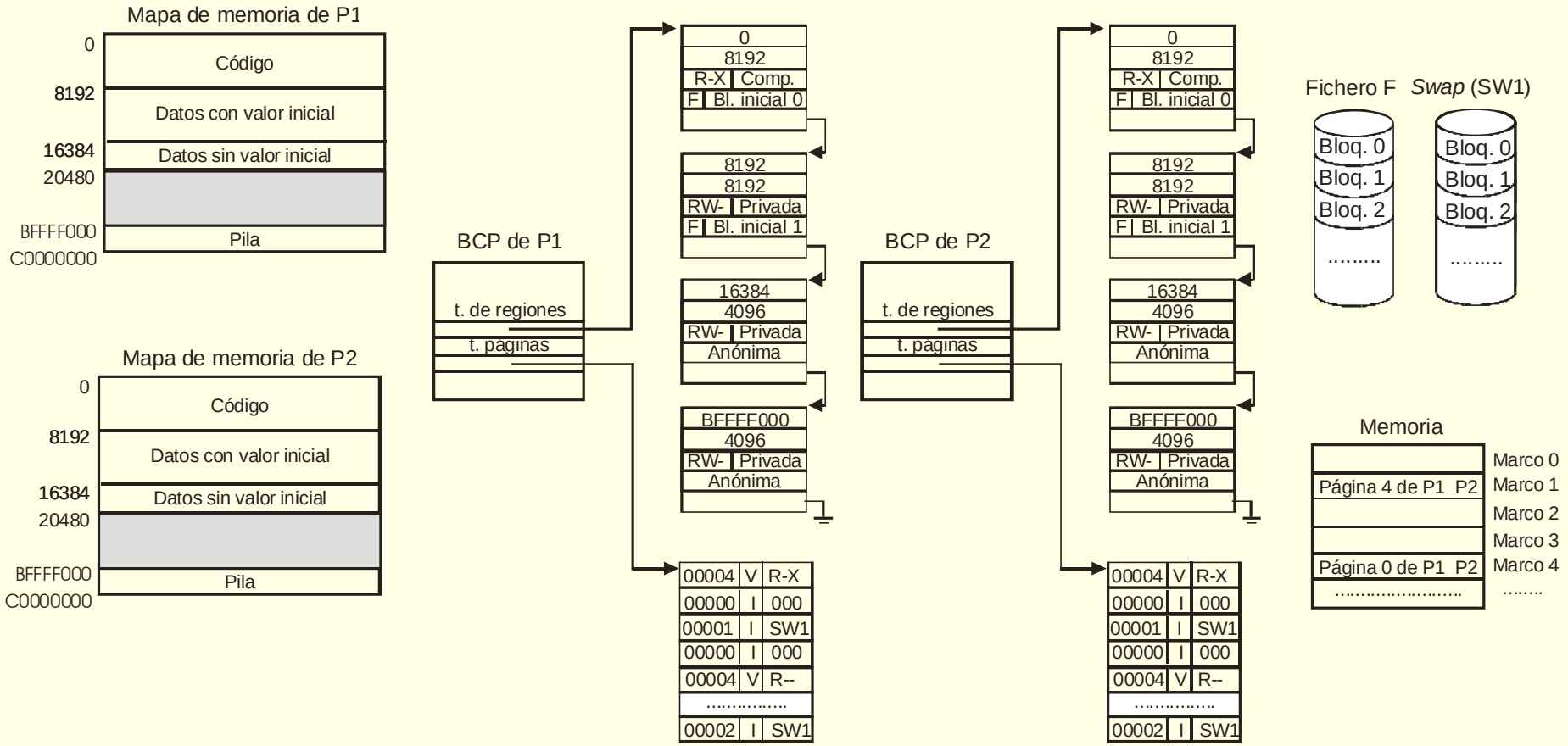


# Expansión de la región de pila

- Expansión automática: programa ↓SP y accede → fallo página
- Extensión de tratamiento de fallo de página:
  - Si dirección inválida (∉ región)
    - Si dirección < SP –› Aborta proceso o le manda señal
    - Si no –› Expansión de pila
- Sutileza:
  - Al menos 1 página inválida entre pila y región más cercana

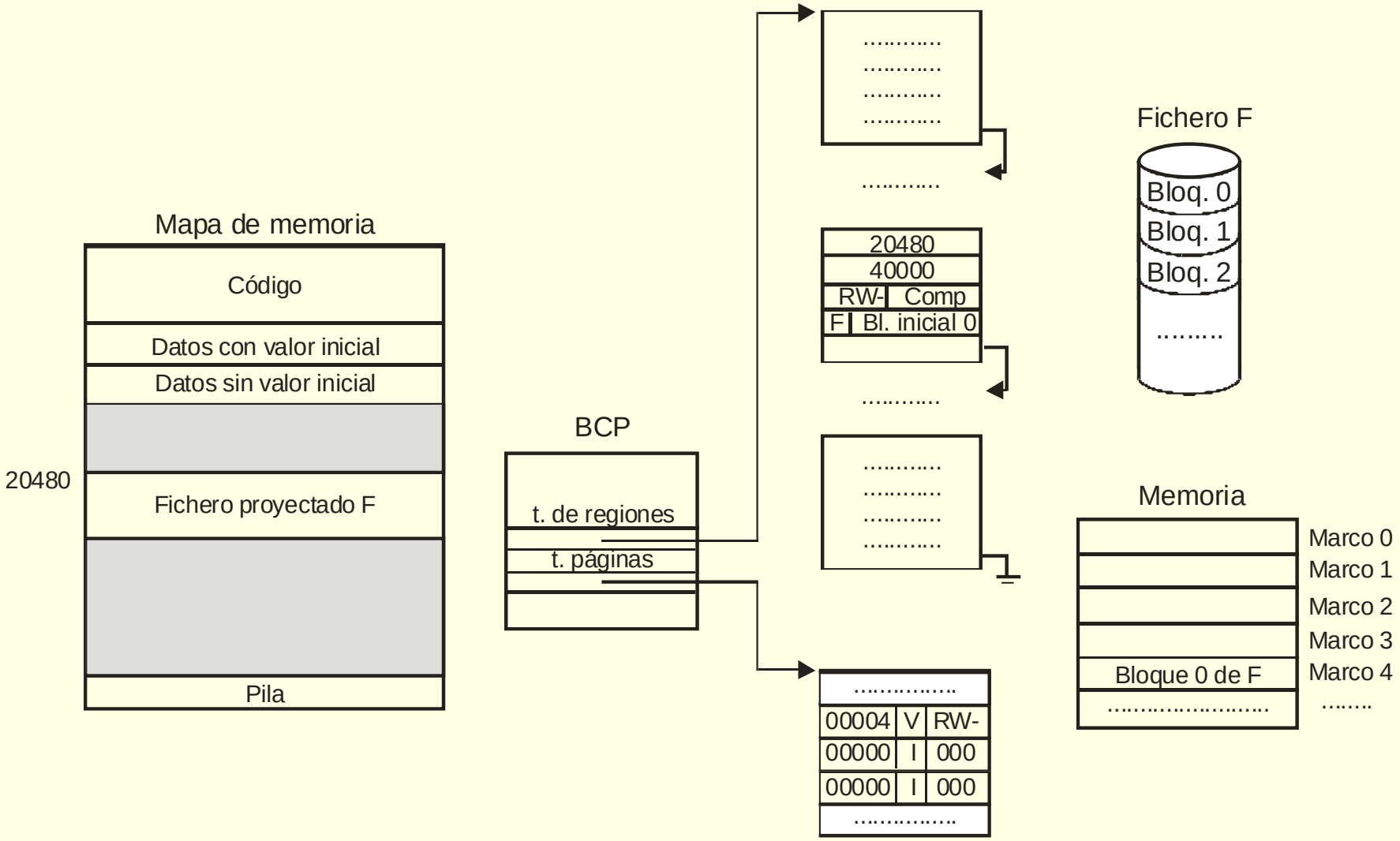
# Resultado del `fork`



# Ficheros proyectados en memoria

- Generalización de memoria virtual
  - Entradas de TP referencian a un fichero de usuario
- Programa solicita proyección de fichero (o parte) en su mapa
  - Puede especificar protección y si privada o compartida
- Programa accede a posición de memoria asociada a fichero
  - Está accediendo al fichero
- Forma alternativa de acceso a ficheros frente a *read*/*write*
  - Menos llamadas al sistema
  - Se evitan copias intermedias en caché de sistema de ficheros
  - Se facilita programación: acceso a estructuras de datos

# Proyección de un fichero



# Problemas de coherencia en jerarquía de memoria

- Entre caché y memoria en E/S por DMA: Si no resuelto por MMU
  - SO realiza volcados e invalidaciones necesarias
- Entre caché virtual y memoria:
  - SO invalida entradas en caché virtual asociadas a una página
    - Si cambia marco asociado a página o se elimina página del mapa
- En la jerarquía de traducción de páginas: TLB y TP
  - SO actualiza TP→ SO debe invalidar entrada TLB
    - Si cambia marco asociado a página o se elimina página del mapa
    - O si hay cambios en cualquier otro campo: protección, ref, mod,
- Por caché de I/D separadas: Si no resuelto por MMU
  - SO realiza volcados e invalidaciones necesarias
- Entre múltiples cachés en multiproc. → resuelto por MMUs
- Entre múltiples TLB en multiproc → SO responsable
  - Propagar invalidación de TLB (uso de IPI)

# Problemas por homónimos y sinónimos

- Homónimos: mismo nombre para objetos diferentes
  - Misma dirección lógica de distintos procesos
  - En TLB y caché virtual si no incluyen PID
    - En cada cambio de contexto invalidar TLB y caché virtual
    - No requerido si segmentación paginada global o SASOS
- Sinónimos: distintos nombres para mismo objeto (*virtual aliasing*)
  - Problema si un nivel no los detecta y gestiona 2 copias
  - Puede darse en caché virtual aunque incluya PID:
    - Si no resuelto por hardware, distintas soluciones:
    - En cambio de contexto, invalidar entradas conflictivas
    - Restricciones de alineamiento en regiones compartidas
      - Asegurar distintos nombres estén asociados a misma línea de caché
    - Uso de Segmentación paginada global o SASOS → No sinónimos
- *https://www.kernel.org/doc/Documentation/cachetlb.txt*